SE | RWTH AACHEN UNIVERSITY

Timo Greifenberg

Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte

Aachener Informatik-Berichte,
Software Engineering

Band 42

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Timo Greifenberg, M.Sc. RWTH
aus Aachen

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr.-Ing. Steffen Becker

Tag der mündlichen Prüfung: 23. Mai 2019

[Gre19] T. Greifenberg:
Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte.
Shaker Verlag, ISBN 978-3-8440-6879-5. Aachener Informatik-Berichte, Software Engineering, Band 42. August 2019.
www.se-rwth.de/publications/

Kurzfassung

Durch den Einsatz von Modellen lässt sich bei der modellgetriebenen Softwareentwicklung (engl. *Model-Driven Development* (MDD)) zum einen die Komplexität des zu entwickelnden Softwareprodukts beherrschen und zum anderen die konzeptionelle Lücke zwischen Problem- und Lösungsdomäne schließen. Die Verwendung und agile Weiterentwicklung von MDD-Werkzeugen kann jedoch selbst zu einem Anstieg der Komplexität im Projekt führen. Durch die große Anzahl an verschiedenen Artefakten und Beziehungen, den verschiedenen Artefakt- und Beziehungsarten und dem hohen Grad der Automatisierung werden das Verständnis, die Wartung und die Weiterentwicklung von MDD-Projekten erschwert. Um dieser Komplexität zu begegnen, ist eine systematische Herangehensweise für das Verständnis und die Verwaltung von MDD-Projekten mit Fokus auf den Artefakten und deren Beziehungen notwendig. In dieser Arbeit wird mit der *artefaktbasierten Analyse* ein solcher Ansatz zur Analyse von Artefaktstrukturen in MDD-Projekten vorgestellt. Der Ansatz sieht den Einsatz einer adäquaten Modellierungstechnik und die Verwendung von Analysewerkzeugen zur Durchführung von Analysen unter Verwendung zugehöriger Methodiken vor. Die Ergebnisse der Arbeit lassen sich wie folgt zusammenfassen:

- Entwicklung einer Modellierungstechnik basierend auf den UML-Sprachen Klassendiagramme, Object Constraint Language und Objektdiagramme zur präzisen Beschreibung möglicher Projektsituationen und der aktuellen Projektsituation sowie zur Spezifikation von Analysen.
- Modellierung eines konkreten *Artefaktmodells* zur Darstellung der Artefakt- und Beziehungsarten MontiCore-basierter MDD-Projekte sowie zugehöriger beispielhafter Analysen.
- Entwicklung einer Gesamtmethodik bestehend aus einer Teilmethodik zur Erstellung projektspezifischer Artefaktmodelle, einer Teilmethodik zur Spezifikation automatisch ausführbarer Analysen und einem Vorgehen zur Durchführung von artefaktbasierten Analysen.
- Entwicklung einer Werkzeugkette zur Unterstützung der Analysen sowie einer Methodik zur Verwendung, Konfiguration, Anpassung und Integration der Werkzeugkette.

Die Durchführbarkeit des Ansatzes wurde an realen Projekten des Lehrstuhls für Software Engineering der RWTH Aachen University erprobt, wodurch Optimierungspotenziale dieser

Projekte aufgezeigt werden konnten. Diese Arbeit trägt dazu bei, die steigenden Komplexität großer MDD-Projekte durch den Einsatz von artefaktbasierten Analysen unter Verwendung adäquater Modellierungstechniken, Werkzeuge und Methoden beherrschbar zu machen.

Abstract

The usage of models within model-driven software development facilitates managing the complexity of the system under development and closes the gap between the problem and the solution domain. The usage and agile development of *Model-Driven Development* tools can itself increase the complexity within a project. The huge number of different artifacts and relations, the different artifact and relation kinds and the high degree of automation hinders the understanding, maintenance and evolution within MDD projects.

A systematic approach to understand and manage MDD projects with a focus on its artifacts and corresponding relations is necessary to handle the complexity. This thesis presents such an approach for the analysis of artifact structures within MDD projects. The approach relies on appropriate modeling techniques and intends the usage of analysis tools to perform analyses using corresponding methodologies. The results of this thesis can be summarized as follows:

- Development of a modeling technique based on the UML languages Class Diagrams, Object Constraint Language and Object Diagrams for the precise definition of possible project situations and the actual project situation as well as for the specification of analyses.
- Modeling of a concrete *artifact model*, which defines the artifacts and relation kinds within MontiCore-based MDD projects. Related exemplary analyses are given in addition.
- Development of an overall methodology consisting of a methodology to create project specific artifact models, a methodology for the specification of automated performed analyses and a methodology for the conduct of artifact-based analyses.
- Development of a tool chain supporting artifact-based analyses and a methodology for its usage, configuration, adaption and integration.

The application of the approach has been demonstrated in real projects of the Chair of Software Engineering of RWTH Aachen University, which led to the investigation of optimization potential of those projects. This thesis contributes to handling the growing complexity in big MDD projects by applying artifact-based analyses relying on appropriate modeling techniques, tools and methodologies.

Danksagung

Ich bedanke mich bei all denen, die mich während meiner Promotion begleitet, unterstützt, mir den nötigen Rückhalt gegeben oder für passende Ablenkungen gesorgt und dadurch zu dem Erfolg meiner Promotion beigetragen haben. Mein besonderer Dank gilt Prof. Dr. Bernhard Rumpe, der mir die Möglichkeit der Promotion am Lehrstuhl für Software Engineering der RWTH Aachen gegeben hat. Insbesondere danke ich ihm für seine Betreuung, seine Ratschläge und das Feedback zu meiner Arbeit. Das Weitern möchte ich Prof. Dr. Steffen Becker für die Bereitschaft zur Übernahme der Zweitkorrektur danken. Auch Prof. Dr. Gerhard Woeginger möchte ich für die Leitung sowie Prof. Dr. Thomas Noll für die Mitarbeit in meinem Prüfungskomitee danken.

Herzlich bedanken möchte ich mich auch bei allen Kollegen, die mich während und teils auch außerhalb meiner Zeit am Lehrstuhl begleitet haben, von denen ich viel lernen durfte und deren Zusammenarbeit ich sehr geschätzt habe. Außerdem bedanke ich mich bei allen Studenten, die ich betreuen durfte und die durch ihre Abschlussarbeiten tatkräftig zur Entstehung meiner Arbeit beigetragen haben. Weiterhin bedanke ich mich bei Marita Breuer, Robert Eikermann, Sylvia Gunder und Galina Volkova, die mir in meiner Zeit am Lehrstuhl organisatorisch und durch Wartung und Bereitstellung der benötigten Infrastruktur sehr geholfen haben. Besonders möchte ich mich auch bei allen wissenschaftlichen Mitarbeitern bedanken, die mir zu Teilen meiner Arbeit Feedback in Form von Reviews gegeben haben: Kai Adam, Arvid Butting, Imke Drave, Steffen Hillemacher, Dr. Katrin Holldöbler, Robert Eikermann, Oliver Kautz, Dr. Christoph Schulze und Dr. Andreas Wortmann.

Besonderer Dank gilt auch meinen Eltern Susanne und Jürgen. Ihr habt mich immer sehr unterstützt, und mir den notwendigen Rückhalt gegeben und dadurch die Promotion überhaupt erst ermöglicht. Außerdem möchte ich mich bei meiner Schwester Anja und bei den Familien Boshe-Plois, Hammes, Kolks, van Reimersdahl und Weber bedanken. Für mich gehört ihr alle zu meiner Familie, wart mir immer eine Stütze und habt mir auch für schwierige Aufgaben Kraft gegeben. Schließlich möchte ich mich noch ganz besonders bei meiner Freundin Nadja bedanken. Ich konnte mich stets auf deine Unterstützung verlassen und weiß, wie schwer es dir gefallen ist, auf die gemeinsame Zeit zu verzichten. Du hast mich motiviert und mir immer den Rücken freigehalten, wodurch ich mich mit der notwendigen Intensität der Promotion widmen konnte. Danke!

Aachen, Juli 2019
Timo Greifenberg

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Kontext der Arbeit 3
1.2. Ziele und Ergebnisse der Arbeit 4
1.3. Aufbau der Arbeit 5

2. Grundlagen modellgetriebener Softwareentwicklungsprojekte **7**

2.1. Modellbasierte und modellgetriebene Softwareentwicklung 7
2.1.1. Sprach- und Modellkomposition 8
2.1.2. Modelltransformationen 9
2.2. Modellgetriebene Softwareentwicklungsprojekte 10
2.3. MontiCore 12
2.3.1. Überblick 12
2.3.2. Symboltabelle 13
2.3.3. Templates 14
2.3.4. Reports 15
2.4. Ausgewählte Modellierungssprachen und modellverarbeitende Werkzeuge . 16
2.4.1. UML/P 16
2.4.2. MontiArc 18
2.4.3. DEx 19

3. Analyse von Softwareentwicklungsprojekten **21**

3.1. Software Reverse Engineering 21
3.2. Traceability und Artefaktabhängigkeiten 22
3.3. Extraktion von Traceabilityinformationen 25
3.3.1. Statische Extraktion 25
3.3.2. Dynamische Extraktion 26
3.4. Nutzung von Traceabilityinformationen 27
3.4.1. Abhängigkeitsanalyse 27
3.4.2. Change-Impact-Analyse 27
3.4.3. Abdeckungsanalyse 28
3.5. Architekturrekonstruktion und Architekturkonformität 28
3.6. Metriken in der Softwareentwicklung 30

3.7. Visualisierung von Abhängigkeiten in der Softwareentwicklung 32
3.7.1. Abhängigkeitsmatrix 32
3.7.2. Graphdarstellung 33
3.7.3. Softwarestadt 34
3.7.4. Architekturvisualisierung 35

4. Artefaktbasierte Analyse 39
4.1. Artefakte in modellgetriebenen Softwareentwicklungsprojekten 39
4.2. Szenario 41
4.3. Lösungsansatz 44
4.4. Grundlegende Begriffe 47
4.5. Anforderungen 48
4.5.1. Anforderungen an Analyseverfahren 49
4.5.2. Anforderungen an die Modellierungstechnik 49
4.5.3. Anforderungen an die Werkzeugunterstützung 50

5. Modellierungstechnik für die Artefaktmodellierung 53
5.1. Verwendung der UML/P zur Modellierung von Artefaktzusammenhängen . . 53
5.1.1. Klassendiagramme 54
5.1.2. Objektdiagramme 55
5.1.3. Object Constraint Language 57
5.1.4. Semantik 58
5.2. Artefaktmodellkern 61
5.2.1. Artefakte und Artefaktcontainer 61
5.2.2. Systeme und Werkzeuge 67

6. Artefaktmodell für modellgetriebene Softwareentwicklungsprojekte mit MontiCore 71
6.1. Java Artefakte 71
6.1.1. Java-Quellcode- und .class-Dateien 72
6.1.2. Beziehungen zwischen Java-Artefakten und Java-Typen 75
6.2. Definition von Modellierungssprachen 80
6.2.1. Modellierungssprachen 81
6.2.2. Grammatiken 82
6.2.3. Modelldateien 83
6.3. Klassendiagramm Artefakte 83
6.4. Generatorartefakte 86
6.4.1. Templates 86
6.4.2. Generatoren 88
6.5. Dynamische Beziehungen zur Generierungszeit 90
6.5.1. Aktionen und Events 90

6.5.2. Aktionen des Generierungsprozesses . . . 92
6.5.3. Ein- und Ausgaben von Werkzeugen . . . 94
6.5.4. Beteiligung von Java und Templatedateien an der Generierung . . . 96
6.6. Beispielhafte Analysespezifikationen . . . 97
6.6.1. Architekturrekonstruktion . . . 98
6.6.2. Differenzarchitektur . . . 99
6.6.3. Indirekte Templatebeziehungen . . . 101
6.6.4. Ungenutzte Importe . . . 102
6.6.5. Inkrementelle Ausführung von Werkzeugketten . . . 104
6.7. Verwandte Ansätze . . . 106
6.7.1. Megamodelle . . . 106
6.7.2. Artefaktorientierung und Artefaktmodelle . . . 107

7. Methodik zur artefaktbasierten Analyse modellgetriebener Softwareentwicklungsprojekte 111
7.1. Methodik zur Erstellung projektspezifischer Artefaktmodelle . . . 112
7.2. Methodik zur Spezifikation projektspezifischer Analysen . . . 115
7.2.1. OCL-basierte Analysen . . . 116
7.2.2. Abhängigkeitsanalyse und Change-Impact-Analyse . . . 116
7.2.3. Architekturanalysen . . . 117
7.2.4. Metriken . . . 118
7.2.5. Nutzungsart von OCL-Ausdrücken spezifizieren . . . 120
7.3. Datenextraktion . . . 121
7.4. Merge von Artefaktdaten . . . 123
7.5. Datenprüfung und -anreicherung . . . 127
7.5.1. Auswertung von OCL-Prüfbedingungen . . . 127
7.5.2. Anreicherung . . . 127
7.6. Analyse von Artefaktdaten . . . 129

8. MontiArT: Eine Werkzeugkette zur Durchführung artefaktbasierter Analysen 137
8.1. Werkzeuge . . . 138
8.1.1. Ein- und Ausgaben . . . 139
8.1.2. Nutzung . . . 142
8.1.3. Implementierung . . . 142
8.2. Datenextraktion . . . 145
8.2.1. Aufbau statischer Extraktoren . . . 146
8.2.2. Artifact Container Extractor . . . 148
8.2.3. Template Extractor . . . 150
8.2.4. Java Extractor . . . 151

8.3. Merge von Artefaktdaten . . . 152
8.3.1. OD Merger . . . 152
8.3.2. Verwendung des OD Mergers in MontiArT . . . 154
8.4. Datenprüfung und Anreicherung . . . 154
8.4.1. Fehlerarten bei der Konsistenzprüfung . . . 155
8.4.2. Strukturprüfung . . . 156
8.4.3. OCL Generator . . . 157
8.4.4. Auswertung von OCL-Prüfbedingungen . . . 160
8.4.5. Anreicherung der abgeleiteten Eigenschaften . . . 162
8.5. Datenanalyse . . . 163
8.5.1. Artifact Data Explorer . . . 163
8.5.2. OCL-basierte Analysen . . . 164
8.5.3. OD Filter . . . 165
8.5.4. Architekturanalyse . . . 166
8.5.5. Visualisierung von Artefaktdaten . . . 168
8.6. Konverterwerkzeuge . . . 177
8.6.1. OD Flattener . . . 177
8.6.2. OD2JSON Transformation . . . 178
8.6.3. CD OCL Splitter . . . 179
8.6.4. CD Merger . . . 180

9. Methodik zur Erstellung von MontiArT-Instanzen 183
9.1. Auswahl mitgelieferter Extraktoren . . . 184
9.2. Implementierung projektspezifischer Werkzeuge . . . 184
9.2.1. Implementierung neuer Werkzeuge . . . 185
9.2.2. Extraktion von Artefaktbeziehungen mit Hilfe der Symboltabelle . . . 185
9.2.3. Verwendung und Anpassung der Reportinginfrastruktur . . . 188
9.3. Fehlererkennung bei skriptbasierter Verkettung von Werkzeugen . . . 190
9.4. Konfiguration von ArtViz . . . 192
9.5. Einbinden von MontiArT in eine Software-Engineering-Infrastruktur . . . 196
9.5.1. Software-Engineering-Infrastruktur . . . 196
9.5.2. Einbinden der Werkzeugkette . . . 197

10. Artefaktbasierte Analysen in der Anwendung 199
10.1. Der DEx-Generator . . . 199
10.1.1. DEx Transformation Extractor . . . 200
10.1.2. DEx Action Reporter . . . 202
10.1.3. Durchführung der Analyse . . . 205
10.2. Eingabeartefakte des BumperBots . . . 209
10.3. Make-basierte Buildprozesse . . . 212
10.3.1. Methodik zur Instrumentierung von Makefiles . . . 214

10.3.2. Werkzeug zur automatisierten Instrumentierung von Makefiles 215
10.4. Inkrementelle Generierung mit MontiCore 216
10.4.1. Lösungsansätze 221
10.4.2. Evaluation 225
10.5. Artefakte bei der Testfallgenerierung im Automotivekontext 227

11. Zusammenfassung und Ausblick **231**
11.1. Zusammenfassung 231
11.2. Ausblick 233

Literaturverzeichnis **235**

A. Markierungen in Abbildungen und Listings **257**

B. Tracing der Anforderungen **259**

C. Artefaktmodell für MontiCore-basierte MDD-Projekte **261**

D. Tagschema für artefaktbasierte Analysen **275**

E. MontiArT-Werkzeuge **277**

F. DEx-Artefaktanalyse **279**
F.1. Definition der DEx-MontiArt-Instanz als Skript 279
F.2. Moduldefinition und Soll-Architektur 285
F.3. Ist-Architektur 297
F.4. Differenzarchitektur 300

G. BumperBot-Artefaktanalyse **303**
G.1. Extrahierte Artefaktdaten 303
G.2. Ungenutzte Importe 308

H. Lebenslauf **309**

Abkürzungsverzeichnis **311**

Abbildungsverzeichnis **313**

Listings **317**

Tabellenverzeichnis **321**

Kapitel 1.

Einleitung

Die Komplexität bei der Entwicklung moderner Softwaresysteme bzw. von Systemen mit einem hohen Softwareanteil steigt immer weiter an. In [EF17] wird Software bereits als der wichtigste Faktor für den Wettbewerb in der Automobilindustrie genannt, wodurch dort die Komplexität gerade des Softwareanteils rapide ansteigt. Weitere Beispiele für komplexe Systeme mit einem hohen Softwareanteil sind cloudbasierte Systeme [KRR14], cyberphysische Systeme (CPS) (engl. *Cyber Physical System* (CPS)) [Lee08] oder Internet of Things (IoT) [AIM10]. Die Verwendung einer *General Purpose Language* (GPL) zur Erstellung solcher Systeme ist fehleranfällig und kostenintensiv, und es existiert eine konzeptionelle Lücke zwischen der Problem- und der Lösungsdomäne [FR07]. Aus diesem Grund müssen beispielsweise Anforderungen an die Systeme von der Problemdomäne in die Lösungsdomäne übertragen werden, wodurch zusätzliche Artefakte im Entwicklungsprozess entstehen. Von einigen Entwicklungsprozessen wird zusätzlich ein Tracing von Entwicklungsartefakten gefordert [MHDZ16]. Die Erstellung und Wartung solcher Beziehungen bedeutet wiederum zusätzlichen Aufwand bei der Entwicklung.

Um zum einen die Komplexität der Software beherrschen zu können und zum anderen die konzeptionelle Lücke zwischen Problem und Lösungsdomäne zu schließen, lassen sich Techniken aus dem Bereich der modellgetriebenen Softwareentwicklung (engl. *Model-Driven Development* (MDD)) einsetzen. Dabei lässt sich durch die Erstellung von plattformunabhängigen Modellen (engl. *Platform Independent Model* (PIM)) [OMG14a] von technischen Sachverhalten der Lösungsdomäne abstrahieren, wodurch die Komplexität besser beherrschbar wird. Solche Modelle der Problemdomäne lassen sich oftmals direkt durch Domänenexperten erstellen. Dabei können für die verschiedenen Sachverhalte des Systems unterschiedliche domänenspezifische Modellierungssprachen (engl. *Domain Specific Language* (DSL)) verwendet werden, wodurch bei der Modellierung auf einzelne Aspekte des Systems fokussiert werden kann [CCF+15b]. Die entstehenden Modelle können anschließend als primäre Eingaben für eine MDD-Werkzeugkette genutzt werden. Solche Werkzeugketten können die Konformität der unterschiedlichen Modelle (sowohl einzelner Modelle als auch der Modelle zueinander) überprüfen, diese analysieren und aus ihnen automatisiert Teile des zu entwickelnden Softwareprodukts erstellen. Durch einen solchen automatisierten MDD-Buildprozess wird die manuelle Synchronisation der Informationen aus Modellen und Quellcode überflüssig, da nur

die Modelle angepasst werden müssen und der zugehörige Quellcode sich ohne zusätzlichen Aufwand automatisiert neu erstellen lässt. Somit zielt der Einsatz von MDD-Ansätzen auf einen effizienteren und effektiveren Softwareentwicklungsprozess ab [BCW12], wodurch eine Verringerung des manuellen Aufwandes, eine Verbesserung der Softwarequalität durch systematische Übersetzung modellierter Sachverhalte in den Quellcode und eine Reduktion der Entwicklungskosten erreicht werden kann.

In MDD-Projekten existieren nicht nur eine große Menge verschiedener Artefakte, was durch die Komplexität der zur erstellenden Software begründet ist, sondern auch eine große Anzahl verschiedener Artefaktarten. Zusätzlich zu den dem Autor aus herkömmlichen Entwicklungsprozessen bekannten Artefaktarten werden in MDD-Projekten weitere Artefaktarten verwendet. Beispiele sind Modelle, Templates und Grammatikdateien, die durch den Einsatz spezieller Technologien zur Erstellung von MDD-Werkzeugen benötigt werden. Auch zwischen diesen spezifischen Arten von Artefakten bestehen heterogene und komplexe Beziehungen, deren Verständnis für die Arbeit von MDD-Werkzeugentwicklern zwingend erforderlich ist. Beispiele für solche Beziehungen sind, dass sich Modellartefakte gegenseitig importieren, ein Artefakt basierend auf Informationen aus einer Menge von anderen Artefakten automatisch generiert wurde oder Artefakte weitere Artefakte enthalten.

Die Anzahl und die Komplexität dieser Abhängigkeiten führen zu einer Reihe neuer Herausforderungen für MDD-Projekte [GHR17]: (1) Schlechte Wartbarkeit wegen unnötigen oder unvorhergesehenen Artefaktbeziehungen bei notwendigen Änderungen, (2) ineffiziente Buildprozesse, die unnötige Prozessschritte ausführen oder nicht inkrementell ausgeführt werden können, (3) lange Entwicklungszeiten, da schlecht organisierte Artefaktabhängigkeiten eine Fehlerquelle darstellen und (4) die Verhinderung der Wiederverwendbarkeit einzelner Werkzeugkomponenten oder von Teilen des generierten Produkts durch unnötige Artefaktbeziehungen. Durch die große Anzahl an verschiedenen Artefakten und Beziehungen, die verschiedenen Artefakt- und Beziehungsarten, die Verwendung von speziellen MDD-Werkzeugen und den hohen Grad der Automatisierung ist ein systematischer Ansatz zur Beherrschung der Komplexität in MDD-Projekten notwendig. Für bestehende Projekte, bei denen kein solcher Ansatz genutzt wurde, können mit Hilfe von Reverse-Engineering-Techniken die Struktur der Artefakte und deren Beziehungen rekonstruiert werden, wodurch das Verständnis der Abhängigkeiten sowie die Analyse von Prozessoptimierungspotenzialen ermöglicht wird. Solche Analysen der Artefakte und deren Beziehungen werden in dieser Arbeit als artefaktbasierte Analysen bezeichnet.

Das Ziel dieser Arbeit ist die Entwicklung von Methoden und Konzepten zur artefaktbasierten Analyse von modellgetriebenen Softwareentwicklungsprojekten. Dazu werden die folgenden Ergebnisse präsentiert:

- Eine Modellierungstechnik zur Modellierung von Artefaktzusammenhängen und zur Spezifikation von zugehöriger Analysen.
- Ein konkretes Modell, welches Artefaktzusammenhänge in MontiCore-basierten Entwicklungsprojekten beschreibt.

- Eine Methodik zur Verwendung der Modellierungstechnik.
- Eine Werkzeugkette, welche die Durchführung von artefaktbasierten Analysen unterstützt.

Im Rest dieses Kapitels wird zunächst in Abschnitt 1.1 der Kontext dieser Arbeit beschrieben. Anschließend werden die wichtigsten Ziele und Beiträge der Arbeit in Abschnitt 1.2 diskutiert, bevor in Abschnitt 1.3 der Aufbau der Arbeit erläutert wird.

1.1. Kontext der Arbeit

Die Grundlage für den entwickelten Ansatz bilden vor allem Vorarbeiten des Lehrstuhls für Software Engineering der RWTH Aachen University. Die zur Modellierung verwendeten Sprachen sind aus der *Unified Modeling Language usable for Programming* (UML/P) [Sch12, Rum16] entnommen. Die Sprachen wurden dabei um zusätzliche Bedingungen erweitert und somit für den Verwendungszweck dieser Arbeit zugeschnitten. Dies ermöglicht eine präzisere Modellierung durch Fokussierung auf die relevanten Modellierungskonzepte sowie eine einfachere Implementierung unterstützender Werkzeuge. Des Weiteren wird eine Taggingsprache zur Spezifikation zusätzlicher Informationen für *Object Constraint Language* (OCL)-Bedingungen genutzt. Die Grundlage für solche Taggingsprachen wurde bereits in [Loo17] durch eine Taggingsprache für Klassendiagramme (engl. *Class Diagram* (CD)) und der zugehörigen Infrastruktur entwickelt.

Die in dieser Arbeit entwickelten Werkzeuge basieren auf der Language Workbench MontiCore [KRV10, HR17]. Darüber hinaus wird der *Data Explorer* (DEx)-Generator [Rot17], ein spezieller, auf MontiCore-basierender Generator, eingesetzt. Dabei verwenden einige der entwickelten Werkzeuge DEx als Unterprogramm während andere Werkzeuge mit Hilfe dieses Generators erstellt wurden.

Die im Rahmen dieser Arbeit analysierten Projekte sind zum Teil ebenfalls in Vorarbeiten entstanden. Zum einen werden DEx-basierte Projekte selbst untersucht, zum anderen werden Projekte untersucht in denen die Sprachfamilie *MontiArcAutomaton* (MAA) [Wor16] und zugehörige Werkzeuge eingesetzt werden. Ein Teil der Methodik zur Analyse von Modellbeziehungen in MontiCore-basierten Projekten nutzt mit der Symboltabelle sowie der zugehörigen Infrastruktur [MSN17] ebenfalls eine Vorarbeit.

Einige der in dieser Arbeit vorgestellten Inhalte wurden bereits im Vorfeld veröffentlicht. Bei diesen Veröffentlichungen wurden die beschriebenen Inhalte zu einem überwiegenden Anteil vom Autor der vorliegenden Arbeit als Teil der Promotion erarbeitet. Dies betrifft die Publikationen [GLR17, BGRW17, GHR17, BGRW18]. Darüber hinaus wurden durch studentische Arbeiten Teile der Ansätze und Konzepte der vorliegenden Arbeit evaluiert bzw. die Umsetzbarkeit der Ansätze und die Anwendbarkeit der Konzepte nachgewiesen. Außerdem haben studentische Arbeiten zur Implementierung der in Kapitel 8 vorgestellten Werkzeugkette

beigetragen. Dies betrifft die Arbeiten [Hil14, Mai15, Con16, Ded16, Hil16, Käs16, Sou16, Cuj17].

1.2. Ziele und Ergebnisse der Arbeit

Die Hauptforschungsfrage dieser Arbeit ist:

> *Wie können artefaktbasierte Analysen agile modellgetriebene Softwareentwicklungsprojekte unter Verwendung adäquater Modellierungstechniken, Werkzeuge und Methoden unterstützen?*

Artefaktbasierte Analysen sind Analysen, die die Artefaktebene betrachten, d.h. es wird nach Möglichkeit von den Inhalten der Artefakte abstrahiert und eine globale Sicht auf das zu untersuchende agile und modellgetriebene Softwareentwicklungsprojekt eingenommen. Artefakte und deren Beziehungen werden somit als Quelle für Analysen genutzt.

Artefakte des MDD-Entwicklungsprozesses sind zum einen die Eingaben, Zwischenergebnisse und Ausgaben des Entwicklungsprozesses. Zum anderen werden aber auch die beim automatisierten Buildprozess verwendeten Werkzeuge betrachtet, die ebenfalls aus Artefakten bestehen. Dies ist wichtig, da davon ausgegangen wird, dass im Rahmen von MDD-Projekten die zur Automatisierung des Entwicklungsprozesses notwendigen Werkzeuge agil und zur Laufzeit des Projekts weiterentwickelt werden können. Die Betrachtung von Artefakten beider Ebenen ermöglicht die Durchführung integrierter Analysen.

Um artefaktbasierte Analysen durchführen zu können, bedarf es einer adäquaten Modellierungstechnik. Die Beschreibung von möglichen Projektsituationen durch Definition der verwendeten Artefakt- und Beziehungsarten, die Darstellung der aktuellen Projektsituation und die Spezifikation durchzuführender Analysen müssen dabei unterstützt werden. Werkzeuge können basierend auf den verwendeten Sprachen die Durchführung der spezifizierten Analysen unterstützen, während Methoden das Vorgehen zur Verwendung der Modellierungssprachen, der Durchführung der Analysen und der Verwendung der Werkzeuge festlegen.

Die Ziele dieser Arbeit sind somit, die Definition einer adäquaten Modellierungstechnik, die Entwicklung von Werkzeugen zur Unterstützung der Durchführung artefaktbasierter Analysen sowie die Entwicklung von zugehörigen Methodiken. Die Ergebnisse der Arbeit lassen sich wie folgt zusammenfassen:

- Entwicklung einer Modellierungstechnik basierend auf den Sprachen CD, OCL und Objektdiagrammen (engl. *Object Diagram* (OD)) zur präzisen Beschreibung möglicher Projektsituationen, der aktuellen Projektsituation sowie der Spezifikation von Analysen. Modelle, die die möglichen Projektsituationen mit Fokus auf den beteiligten Artefakt- und Beziehungsarten beschreiben, werden in dieser Arbeit als *Artefaktmodell* (AM) bezeichnet.

- Modellierung eines konkreten AMs für MontiCore-basierte MDD-Projekte sowie zugehöriger beispielhafter Analysen. Dieses Modell dient somit als Grundlage für die Analyse weiterer MontiCore-basierter Projekte und als Beispiel für die Erstellung eines AMs sowie zugehöriger Analysen basierend auf der entwickelten Methodik (siehe nächsten Punkt). Darüber hinaus lassen sich Teile des AMs auch in anderen Kontexten wiederverwenden (z.B. die Beschreibung von Java-Artefakten und deren Beziehungen).

- Entwicklung einer Methodik zur Anwendung des Ansatzes. Diese Methodik umfasst Teile zur:
  - Erstellung projektspezifischer AMe basierend auf einem wiederverwendbaren AM-Kern.
  - Erstellung projektspezifischer Analysen.
  - Durchführung von artefaktbasierten Analysen.

- Entwicklung einer Werkzeugkette zur Unterstützung der Durchführung von artefaktbasierten Analysen basierend auf der entwickelten Methodik. Die dabei entwickelten MDD-Werkzeuge lassen sich teilweise auch für andere Kontexte außerhalb von artefaktbasierten Analysen nutzen.

- Entwicklung einer Methodik zur Verwendung, Konfiguration, Anpassung und Integration der Werkzeugkette.

- Anwendung der Methodik unter Verwendung der Werkzeugkette, um die Durchführbarkeit des Ansatzes zu demonstrieren. Dabei wurden reale Projekte des Lehrstuhls untersucht und Optimierungspotenziale dieser Projekte aufgezeigt.

1.3. Aufbau der Arbeit

In dieser Arbeit werden graphische Diagramme, textuelle Modelle und Listings mit Programmquellcode oder textuellen Modellen gezeigt. Die verwendete (Modellierungs-) Sprache wird dabei durch entsprechende Markierungen in den Abbildungen und Listings dargestellt. Darüber hinaus können die gleichen Sprachen zur Darstellung von Inhalten verschiedener Art genutzt werden. Um bei Angabe einer Sprache zusätzlich die Art des dargestellten Inhalts in Abbildungen oder Listings zu kennzeichnen, werden Stereotypen unterhalb der Markierungen verwendet. Eine vollständige Liste der verwendeten Markierungen und Stereotypen sowie deren Erklärung lässt sich Anhang A entnehmen. Die weiteren Kapitel der Arbeit sind wie folgt strukturiert:

Kapitel 2 beschreibt als Grundlage dieser Arbeit zentrale Begriffe aus dem Bereich der modellgetriebenen Softwareentwicklung. Darüber hinaus werden relevante Vorarbeiten im

Hinblick auf Konzepte, Modellierungssprachen und Werkzeuge vorgestellt.

Kapitel 3 geht auf Analysen im Bereich der Softwareentwicklung ein. Die dort vorgestellten Analysetechniken bilden Grundlagen und Anforderungen für die in dieser Arbeit entwickelten artefaktbasierten Analysen für MDD-Projekte.

Kapitel 4 definiert die zentralen Begriffe dieser Arbeit: *Artefakt*, *Artefaktmodell*, *Artefaktdaten* und *artefaktbasierte Analyse*. Darüber hinaus wird ein beispielhaftes Anwendungsszenario beschrieben, das die Arbeit weiter motiviert. Außerdem wird ein Überblick über den verwendeten Lösungsansatz gegeben und die an dem Ansatz beteiligten Rollen sowie Anforderungen an den Lösungsansatz vorgestellt.

Kapitel 5 definiert die verwendete Modellierungstechnik durch Angabe der zu verwendenden Sprachen inklusive zusätzlicher einschränkender Bedingungen an einzelne Sprachen sowie deren Komposition. Mit Hilfe dieser Sprachen wird ein wiederverwendbarer AM-Kern spezifiziert, der als zentraler Bestandteil der Modellierungstechnik zur Erstellung von projektspezifischen AMen dient.

Kapitel 6 definiert ein konkretes AM zur Analyse von MontiCore-basierten MDD-Projekten. Weiterhin werden Spezifikationen beispielhafter artefaktbasierter Analysen vorgestellt und es wird auf verwandte Modellierungsansätze eingegangen.

Kapitel 7 stellt eine Methodik zur Erstellung der für die artefaktbasierte Analyse notwendigen Artefakte vor. Diese umfasst zum einen eine Teilmethodik zur Erstellung von projektspezifischen AMen, die aus den Erfahrungen bei der Spezifikation des im vorherigen Kapitels gezeigten AMs entstanden ist. Zum anderen wird eine Teilmethodik zur Spezifikation von zugehörigen Analysen definiert. Außerdem wird das Vorgehen bei der Durchführung artefaktbasierter Analysen vorgestellt.

Kapitel 8 stellt die entwickelte Werkzeugkette zur Unterstützung der Durchführung artefaktbasierter Analysen von MDD-Projekten vor.

Kapitel 9 beschreibt eine Methodik zur Verwendung, Konfiguration und projektspezifischer Anpassung der Werkzeugkette. Außerdem wird die Integration der Werkzeugkette in eine vorhandene Software Engineering Infrastruktur vorgestellt.

Kapitel 10 stellt reale Anwendungen von artefaktbasierten Analysen vor. Neben der Durchführung der Analysen werden außerdem deren Ergebnisse vorgestellt.

Kapitel 11 schließt die Arbeit mit einer Zusammenfassung der Ergebnisse ab und diskutiert mögliche weiterführende und aufbauende Arbeiten.

Kapitel 2.

Grundlagen modellgetriebener Softwareentwicklungsprojekte

In diesem Kapitel werden Grundlagen modellgetriebener Softwareentwicklungsprojekte erläutert. Modellgetriebe Softwareentwicklungsprojekte sind Untersuchungsgegenstand der in dieser Arbeit vorgestellten Analysemethodik, weshalb ein detailliertes Verständnis der dort vorhandenen Vorgänge und verwandten Begrifflichkeiten benötigt wird. Die hier vorgestellten Grundlagen sollen dieses Verständnis vermitteln und umfassen sowohl Definitionen aus den Bereichen modellbasierter und modellgetriebener Softwareentwicklung als auch eine Definition des Projektbegriffs. Darüber hinaus wird besonders auf einige Konzepte, Sprachen und Werkzeuge aus diesem Bereich eingegangen. Diese Grundlagen werden im Verlauf der Arbeit zur Definition der Modellierungstechnik und zur Implementierung der Analysefunktionalität genutzt, sind aber teilweise auch Bestandteile der zu untersuchenden Projekte (siehe Kapitel 10). Grundlagen aus dem Bereich der Softwareanalyse folgen anschließend in Kapitel 3.

2.1. Modellbasierte und modellgetriebene Softwareentwicklung

Dieser Abschnitt führt relevante Begriffe aus der modellbasierten und modellgetriebenen Softwareentwicklung ein. Dabei wurde sich an entsprechenden Abschnitten aus den Arbeiten [Sch12, Wor16] orientiert und Teile daraus übernommen.

In der modellbasierten Softwareentwicklung sind Modelle wichtige Entwicklungsartefakte. Durch Abstraktion mit Hilfe von Modellen lässt sich die steigende Komplexität in modernen Softwareprojekten bewältigen. Modelle können dabei in mehreren Entwicklungsphasen wie beispielsweise Anforderungsspezifikation, Entwicklung oder Deployment eingesetzt werden, um die dortigen Sachverhalte zu beschreiben. Dabei wird von irrelevanten Informationen abstrahiert, sodass nur die relevanten Sachverhalte übrig bleiben. Somit führt die Verwendung von Modellen zu einem leichteren Verständnis der modellierten Aspekte. In der Informatik ist der Modellbegriff nicht eindeutig definiert, stattdessen existieren mehrere Definitionen:

- *„Ein Modell ist seinem Wesen nach eine in Maßstab, Detailliertheit und/oder Funktionalität verkürzte beziehungsweise abstrahierte Darstellung des originalen Systems.“* [Sta73]
- *„A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made.“* [Küh06]
- *„A model is a set of statements about some system under study.“* [Sei03]
- *„A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system.“* [BG01]

In dieser Arbeit wird genau wie in den Arbeiten [Sch12, Wor16] ein Modell anhand der folgenden drei Hauptmerkmale definiert, die bereits in [Sta73] formuliert wurden:

- Abbildungsmerkmal: Basiert auf einem Original.
- Verkürzungsmerkmal: Abstraktion von Eigenschaften des Originals.
- Pragmatisches Merkmal: Schaffung für einen Zweck.

Im Folgenden werden Modelle vor allem in der Entwicklungsphase der Softwareentwicklung betrachtet, wo sich trotz des eingeschränkten Betrachtungsumfangs verschiedene Einsatzmöglichkeiten für Modelle finden lassen. Je nachdem wie formal die verwendeten Modelle spezifiziert werden, lassen sich Modelle automatisiert weiterverarbeiten, sodass Teile oder sogar ganze Implementierungen des zu entwickelnden Softwareprodukts automatisch durch Software-Engineering-Werkzeuge erstellt werden können. Werden Modelle innerhalb der Softwareentwicklung genutzt, beispielsweise als Grundlage für eine manuelle Implementierung, handelt es sich um modellbasierte Softwareentwicklung. Bei der automatischen Verarbeitung der Modelle *treiben* die Modelle den Entwicklungsprozess, weshalb von modellgetriebener Softwareentwicklung gesprochen wird. Die modellgetriebene Softwareentwicklung ist somit ein Spezialfall der modellbasierten Softwareentwicklung. In [Sof17] wird zusätzlich noch zwischen modellgetriebenem Engineering (engl. *Model-Driven Engineering* (MDE)) und modellgetriebener Architektur (engl. *Model-Driven Architecture* (MDA)) [OMG14b] unterschieden, wobei MDE sich auch auf andere Tätigkeiten außerhalb der reinen Entwicklung bezieht und es sich bei der MDA um eine von der *Object Management Group* (OMG) [OMG17b] definierte Ausprägung der modellgetriebenen Softwareentwicklung handelt. Diese Arbeit fokussiert auf die modellgetriebene Softwareentwicklung.

2.1.1. Sprach- und Modellkomposition

Um effektiv große Systeme modellbasiert entwickeln zu können, müssen sich verschiedene Aspekte in verschiedenen Modellartefakten definieren lassen. Dies ermöglicht zum einen

parallele Entwicklungstätigkeiten und zum anderen die Verwendung unterschiedlicher, zu der Problemdomäne passender (oder sogar zugeschnittener) Modellierungssprachen. Damit verschiedene Modellartefakte der gleichen oder verschiedener Sprachen kombiniert verwendet werden können, bedarf es Modell- bzw. Sprachkomposition. Um den Begriff *Globalization of Modeling Language (GEMOC)* [Gem17] ist ein eigener Forschungsbereich zu Fragen der Sprachkomposition entstanden. Im Speziellen wurde dort auch die Komposition von DSLs [CCF+15a] untersucht. Sprachkomposition beschreibt hierbei die integrierte Nutzung verschiedener (Modellierungs-) Sprachen, während Modellkomposition die integrierte Nutzung von Modellen einer oder verschiedener Modellierungssprachen beschreibt. Für die Sprachkomposition existieren verschiedene Ansätze, die auch Auswirkungen auf die Integration zugehöriger Modelle haben. Im Folgenden werden die verschiedenen Möglichkeiten der Sprachkomposition nach [HLMSN+15a, HLMSN+15b] betrachtet.

Sprachaggregation bietet die Möglichkeit mehrere Modellierungssprachen nebeneinander nutzen zu können. Zugehörige Artefakte können miteinander in Beziehung stehen, gehören jedoch immer genau einer der aggregierten Modellierungssprachen an. Aggregierte Modellierungssprachen werden auch Sprachfamilie genannt.

Spracheinbettung ermöglicht die Wiederverwendung einer Sprache an bestimmten Erweiterungspunkten einer anderen Sprache. Die erste Sprache wird somit in die zweite Sprache eingebettet, wodurch eine neue Modellierungssprache entsteht. Modellartefakte gehören dann dieser neuen Sprache an und können Elemente beider Ausgangssprachen enthalten.

Sprachvererbung ist die dritte Möglichkeit der Sprachkomposition. Hierbei erbt eine Sprache von einer anderen Sprache, wodurch sie Konzepte dieser Sprache erweitern, aber auch verändern kann. Somit lässt sich die existierende Sprache zur Definition der neuen Sprache wiederverwenden.

In dieser Arbeit wird vor allem die Modellkomposition betrachtet. Modellartefakte stehen hier durch Sprachaggregation ihrer zugehörigen Sprachen auf verschiedene Art und Weisen miteinander in Beziehung. Bei einer großen Menge an Modellen, zum Beispiel in verteilten Entwicklungsprozessen von komplexen CPS, kann dies zu chaotischen Zuständen in zugehörigen Entwicklungsprojekten führen. Als Sonderfall der Modellkomposition wird aber auch die Komposition von Grammatiken betrachtet. In diesem Zusammenhang lassen sich Grammatiken als Sprachmodelle ansehen.

2.1.2. Modelltransformationen

Modelltransformationen bilden einen wichtigen Teil der modellgetriebenen Softwareentwicklung. Modelltransformationen bekommen Modelle als Eingabe und transformieren diese. Dabei lassen sich Modelltransformationen in verschiedene Kategorien unterteilen. Die beiden Hauptkategorien nach [CH03] bilden Modell-zu-Text (engl. *Model-to-Text* (M2T)) und Modell-zu-Modell (engl. *Model-to-Model* (M2M)) -Transformationen.

Der Unterschied zwischen den beiden Kategorien besteht darin, dass M2M-Transformationen als Ausgabe wiederum ein Modell produzieren, welches häufig ein Zwischenergebnis auf dem

Weg zur Implementierung des Zielprodukts darstellt. Die Ziele solcher Transformationen können vielfältig sein. (1) Modelle können in eine andere Modellierungssprache übersetzt werden, für die bereits ein weiterverarbeitendes Werkzeug existiert, wodurch der Anschluss an eine bestehende Werkzeugkette ermöglicht wird. (2) Modelle können vereinfacht werden, sodass nachfolgende Werkzeuge nicht mit der ganzen Komplexität einer Modellierungssprache konfrontiert werden müssen. In [Wei12] wird beispielsweise gezeigt, wie sich hierarchische Zustandsübergangsdiagramme (engl. *Statechart* (SC)s) durch eine Vortransformation wesentlich einfacher weiterverarbeiten lassen. (3) Modelle können durch zusätzliche Informationen angereichert werden. So wird in der MDA [OMG14a] zwischen plattformunabhängigen (PIM) und plattformspezifischen Modellen (engl. *Platform Specific Model* (PSM)) unterschieden. Sind alle Informationen vorhanden und eine Überführung systematisch durchführbar, ist an dieser Stelle auch eine automatische Transformation per Modelltransformation möglich.

Im Gegensatz zu M2M-Transformationen werden bei M2T-Transformationen Textdateien als Ausgabe produziert. Folgen die Dateien der Syntax einer Programmiersprache, lässt sich im Anschluss ein zugehöriger Compiler verwenden, um ein ausführbares Zielprodukt zu erstellen. Es gibt auch Anwendungsfälle, bei denen durch eine M2T-Transformation keine Quellcodedateien erstellt werden. Denkbar sind hier beispielsweise Konfigurations-, XML-, Tabellen-, oder Reportdateien, welche außerhalb, aber auch als Teil des Zielprodukts Verwendung finden. Wird durch die M2T-Transformation hingegen Quellcode für ein ausführbares System erzeugt, wird das entsprechende Werkzeug auch als Generator und die Transformation als *Modell-zu-Code*-Transformation bezeichnet. Mehrstufige Generierungsprozesse, bei denen ein Generator selbst generiert wird, sind ebenfalls möglich. Eine Übersicht über weitere Unterscheidungskriterien von Modelltransformationssprachen und -werkzeugen lässt sich [CH03, Wei12] entnehmen.

2.2. Modellgetriebene Softwareentwicklungsprojekte

Dieser Abschnitt definiert den Begriff *Modellgetriebenes Softwareentwicklungsprojekt* (MDD-Projekt) im Kontext dieser Arbeit. Dies ist wichtig, da der vorgestellte Ansatz MDD-Projekte als Analysegegenstand nutzt. Es wird eine sehr technische Sichtweise auf solche Projekte angenommen. Der Fokus liegt auf den Entwicklungsartefakten und auf den zur Automatisierung der Implementierungstätigkeit verwendeten Werkzeugen sowie den Beziehungen zwischen Werkzeugen und Artefakten. Außerhalb des Betrachtungsbereiches liegen hingegen weitere Aktivitäten aus dem Bereich des Projektmanagements [PR09]. Diese Aktivitäten können zum Teil eigene Rollen, Artefakte und Werkzeuge mit sich bringen, die ebenfalls miteinander in Beziehung stehen. Aufbauend auf dieser Arbeit könnten solche Aktivitäten mit betrachtet werden. Ferner könnten die verschiedenen Phasen von verwendeten Vorgehens- bzw. Prozessmodellen [LL13] ebenfalls den Betrachtungsumfang erweitern.

In dieser Arbeit wird vor allem die modellgetriebene Entwicklungstätigkeit betrachtet. Hier ist Idee, dass Stakeholder ihre Anforderungen an das zu implementierende System in Form von Modellen formulieren, welche formal genug sind, sodass unter Verwendung von MDD-

Werkzeugen direkt das gewünschte Zielsystem generiert werden kann. Bei dem vorgestellten Ansatz wird auf die vorhandenen Artefakte fokussiert und nicht auf die zugehörigen Tätigkeiten und Rollen. Diese werden zwar im Laufe der Arbeit erörtert, sind jedoch nicht Gegenstand der vorgestellten Analysen. Zusätzlich zu den zu erstellenden Entwicklungsartefakten (Modelle, handgeschriebene Codedateien, Templates, etc.) werden auch die an der (semi-) automatischen Erstellung des Zielsystems beteiligten Werkzeuge, die sich zu Werkzeugketten zusammenschließen lassen, betrachtet. Dabei wird davon ausgegangen, dass Werkzeuge agil und zur Laufzeit des Projekts weiterentwickelt werden, weshalb auch der Aufbau und die Abläufe von Werkzeugen für einzelne sowie integrierte Analysen von Interesse sind. Nach [BJL+15] lassen sich Werkzeuge auf verschiedene Interaktivitätsarten zusammenschließen: dateibasiert, zwischenzeitlicher Zusammenschluss mittels Service API und Echtzeitinteraktion durch Nutzung eines gemeinsamen Modell Repositorys. In dieser Arbeit werden Projekte betrachtet, bei denen die Werkzeugintegration dateibasiert stattfindet. Diese Einschränkung kann getroffen werden, da der Generierungsprozess im Vordergrund der Analyse steht, bei dem Entwicklungsartefakte vor allem dateibasiert vorliegen und automatisiert verarbeitet werden. Bei der Analyse anderer Phasen oder wenn die Verwaltung und nicht die automatische Verarbeitung von Artefakten im Vordergrund steht, könnte Bedarf für die Unterstützung weiterer Interaktivitätsformen entstehen.

Auch der Code des generierten Zielsystems ist Analysegegenstand. Zusätzlich werden die zwischen den genannten Elementen existierenden Beziehungen betrachtet. Diese lassen sich entweder statisch, d.h. durch eine Analyse der Artefaktinhalte, extrahieren oder aber zur Laufzeit des Buildprozesses beobachten. Da die Zusammenhänge auf Artefaktebene in dieser Arbeit im Fokus stehen, werden artefaktinterne Information nur in geringem Maße mit betrachtet. Dies umfasst die Betrachtung von Artefaktinhalten zur Identifikation von Artefaktbeziehungen, jedoch nicht detaillierte Betrachtungen, wie etwa die Berechnung von Testüberdeckungsmetriken [Hof13] oder das Aufspüren von Code Smells [FB99].

Zusammenfassend lässt sich sagen, dass in dieser Arbeit vor allem auf die vorhandenen Artefakte und deren Beziehungen im MDD-Prozess eingegangen wird. Es wird außerdem von einem wiederholbaren Buildprozess ausgegangen, der so weit automatisiert ist, dass keine menschliche Interaktion nach dem Auslösen des Prozesses benötigt wird. Daraus folgt, dass Eingabeartefakte bei Ausführung des Buildprozesses nicht modifiziert und generierte Artefakte anschließend nicht manuell verändert werden dürfen. Außerdem wird von der Verwendung von Mischartefakten aus generierten und handgeschriebenen Anteilen [GHK+15a, GHK+15b] abgesehen, um die Komplexität der MDD-Projekte nicht unnötig zu erhöhen. Analysegegenstand der Projekte sind Eingaben, Zwischenergebnisse, Ausgaben, Werkzeuge und jegliche Art von Beziehungen zwischen den genannten Elementen im MDD-Prozess. Dabei wird von Artefaktinhalten weitestgehend abstrahiert.

Wenn im Folgenden von MDD-Projekt gesprochen wird, wird diese Sichtweise auf solche Projekte angenommen. Darüber hinaus wird im Kontext dieser Arbeit MontiCore als vorgegebene Technologie zur Realisierung von MDD-Projekten angenommen. Es gibt jedoch keinen Grund zur Annahme, dass sich die vorgestellten Konzepte nicht auf durch andere Technologien

realisierte MDD-Projekte übertragen lassen. Der Begriff (MDD-) Buildprozess wird verwendet, wenn die reine Ausführung der MDD-Werkzeugkette gemeint ist. Wird zusätzlich die manuelle Erstellung von Eingabeartefakten sowie die Weiterentwicklung und Konfiguration der MDD-Werkzeugkette betrachtet, so wird der Begriff MDD-Entwicklungsprozess verwendet.

2.3. MontiCore

Die in dieser Arbeit betrachteten MDD-Projekte verwenden auf MontiCore-basierende MDD-Werkzeuge. MontiCore wurde darüber hinaus zur Implementierung der entwickelten Analysewerkzeuge (siehe Kapitel 8) genutzt. Somit stellt MontiCore eine zentrale Vorarbeit dieser Arbeit dar, weshalb der Aufbau sowie die Abläufe von MontiCore-basierten Werkzeugen zum Verständnis der weiteren Arbeit benötigt werden. Im Folgenden werden die benötigten Grundlagen von MontiCore vorgestellt

2.3.1. Überblick

MontiCore [GKR+06, KRV08, GKR+08, KRV10, Kra10, Völ11, HR17] ist eine Language Workbench zur effizienten Entwicklung kompositioneller Modellierungssprachen. Abbildung 2.1 gibt einen Überblick über den Aufbau und den Ablauf von MontiCore-basierten Generatoren.

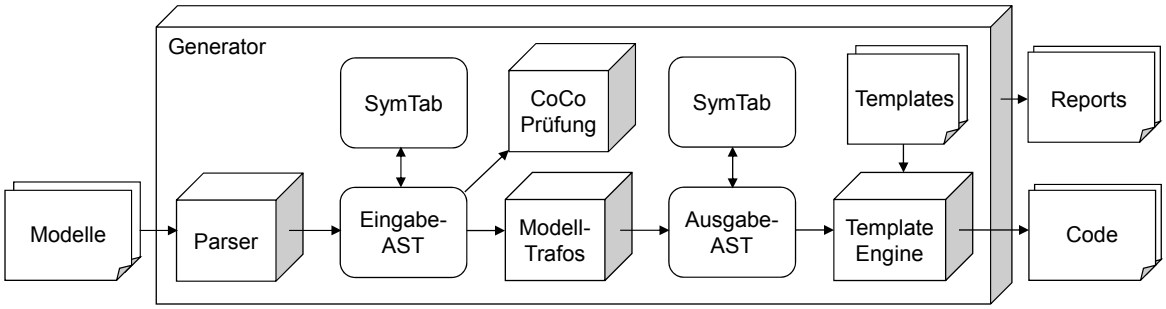

Abbildung 2.1.: Aufbau und Ablauf von MontiCore-basierten Generatoren.

MontiCore-Sprachen werden in erster Linie mittels kontextfreier Grammatiken spezifiziert. Aus einer Grammatik lässt sich unter anderem ein Parser, eine Datenstruktur für abstrakte Syntaxbäume (engl. *Abstract Syntax Tree* (AST)) sowie eine Implementierung des Visitorentwurfsmusters [GHJV95] für die AST-Datenstruktur generieren. Die AST-Datenstruktur lässt sich durch eine Symboltabellendatenstruktur erweitern, welche als Modellschnittstelle fungiert, dynamisch Modellartefakte nachladen kann und sich teilweise generieren lässt [MSN17]. Somit ermöglicht die Symboltabelle die integrierte Verarbeitung mehrerer Modelle verschiedener Modellierungssprachen. Durch die Erstellung von Wohlgeformtheitsregeln, sogenannten Kontextbedingungen (engl.: *Context Condition* (CoCo)), lassen sich zusätzliche

Bedingungen zu den durch die Grammatik vorgegebenen Regeln hinzufügen. Diese Bedingungen werden auf der zusammenhängenden Datenstruktur aus AST und Symboltabelle geprüft. Als vorbereitenden Schritt vor der Generierung können M2M-Transformation verwendet werden. Diese können entweder in Form von Java-Code implementiert oder mit Hilfe spezieller Transformationssprachen [Wei12, Höl18] definiert werden. Eine Methodik zur Erstellung neuer Transformationssprachen steht ebenfalls zur Verfügung [HRW15]. Anschließend lässt sich durch templatebasierte Codegenerierung der Quellcode des zu erstellenden Systems generieren. Als Nebenprodukt werden zusätzlich Berichte (engl. *Report*) erstellt, welche relevante Events des letzten Generierungsdurchlaufs protokollieren oder in informativer Form zusammenfassen.

MontiCore generiert somit eine Infrastruktur, um Modelle zu verarbeiten, sie auf ihre Wohlgeformtheit hin zu prüfen, sie in M2M-Transformationen zu transformieren und schließlich Quellcodeartefakte zu generieren. Darüber hinaus wird MontiCore zusammen mit einer Laufzeitumgebung (engl. *Runtime Environment* (RTE)) ausgeliefert, deren Funktionen und Datenstrukturen sowohl von den generierten als auch von den handgeschriebenen Anteilen MontiCore-basierter Werkzeuge genutzt werden können. MontiCore wurde bereits zur Erstellung von Sprachen und zugehörigen Werkzeugen in verschiedenen Domänen, einschließlich der Automotive [RSW+15] und Robotik [AHRW17] Domänen sowie im Bereich von Cloudanwendungen [NPR13], verwendet. Im Folgenden werden einige für die Arbeit besonders relevante Teile von MontiCore detaillierter eingeführt.

2.3.2. Symboltabelle

Symboltabellen von MontiCore-basierten Sprachen dienen dazu, die relevanten Informationen zu Modellelementen bei der Verarbeitung einzelner Modelle, aber auch über Modellgrenzen hinweg, verfügbar und leicht zugänglich zu machen. Informationen zu Modellelementen können, wenn die verwendete Modellierungssprache entsprechend entworfen wurde, über einen eindeutigen Namen abgerufen werden. Die Informationen werden für solche Modellelemente in gebündelter Form von der Symboltabelle bereitgestellt. Die nachfolgende Definition der Symboltabelle ist [MSN17] entnommen. Sie basiert auf den Arbeiten von [ALSU07, Völ11, HMSNR15b, HLMSN+15a].

Definition 2.3.1 (Symboltabelle (ST))**.**

1. *„The ST is a data structure consisting of a scope graph with an associated collection of symbols at each scope. It maps names to essential information about model elements, represented as symbols. The ST allows to efficiently organize and find, among others, declarations, types, and implementation details associated with those model elements.“*

2. *„Moreover, the ST enables to efficiently navigate between AST nodes of the considered model(s).“*

3. *„Furthermore, the ST represents the essence of a language, i.e., of its models. This especially includes the model interfaces constituted by the language interface (cf. [CBCR15]).“*

Die Einheit der gebündelten Informationen eines benannten Modellelements wird als Symbol bezeichnet. [MSN17] definiert den Begriff Symbol wie folgt:

Definition 2.3.2 (Symbol)**.** *„A symbol definition (or short symbol) contains all essential information about a named model element. It has a specific kind depending on the model element it denotes. A symbol is defined exactly once.“*

Die für die Symboltabelle benötigte Datenstruktur lässt sich in MontiCore teilweise generieren. Anpassungen und Ergänzungen der Standarddatenstruktur können per handgeschriebenem Code erfolgen. Andere Teile der Symboltabelle, wie der Algorithmus zur Initialisierung der Datenstruktur, müssen derzeit für jede Sprache händisch implementiert werden.

Die Symboltabellendatenstruktur und die AST-Datenstruktur sind integrierte Teile einer Gesamtdatenstruktur, wodurch je nach Bedarf zwischen den beiden Substrukturen ausgewählt werden kann oder beide Strukturen integriert genutzt werden können. Genutzt werden die Datenstrukturen vor allem bei der Wohlgeformtheitsprüfung der Eingabemodelle oder bei der templatebasierten Codegenerierung. In der verwendeten Version von MontiCore (4.5.3) unterstützt die AST-Datenstruktur im Gegensatz zur Symboltabellendatenstruktur das Visitorentwurfsmuster [GHJV95], welches eine komfortable Traversierung des ASTs ermöglicht. Demgegenüber stellt die Symboltabelle die Schnittstelle von Modellen dar und ermöglicht ein komfortables Auflösen von Referenzen auf andere Modellelemente. Beim Auflösen der Referenzen können Artefakte dynamisch nachgeladen werden.

In dieser Arbeit wird die Symboltabelle zur Implementierung von benötigten MDD-Werkzeugen verwendet. Darüber hinaus wird in Abschnitt 9.2.2 eine Methodik beschrieben, wie sich die Symboltabelle zur Analyse von Beziehungen zwischen Modellartefakten nutzen lässt.

2.3.3. Templates

MontiCore unterstützt die templatebasierte Generierung [Fow10] mittels der Templateengine FreeMarker [Fre17]. Dabei werden M2T-Transformationen ausgeführt, bei denen Modelle in Quellcode des zu entwickelnden Produkts transformiert werden. Hierbei ist es möglich, dass ein so erstelltes Produkt selbst wieder ein MDD-Werkzeug ist, wie es beispielsweise bei der Verwendung von MontiCore für die Erstellung von Generatoren und auch bei dem MontiCore-Bootstrapping [Kra10] der Fall ist. Darüber hinaus ist aber auch die Erstellung anderer Artefaktarten wie beispielsweise Reports (siehe Abschnitt 2.3.4) denkbar.

Bei der M2T-Transformation wird mittels Templateartefakten Code im Zielformat spezifiziert, welcher variable Anteile enthält. Diese Anteile werden zur Laufzeit basierend auf den Eingaben des Generators befüllt. Fowler beschreibt die Funktionalität von Templates wie folgt: *„The basic idea behind Templated Generation is to write the output file you desire, inserting callouts for all the bits that vary. You then use a template processor with the template file and a context that can fill the callouts to populate the real output file.“* [Fow10] Eine Ähnliche Definition lässt sich auch in [CFJ+16] finden: *„Is a raw piece of an artifact with explicitly*

marked expressions that are evaluated relative to a model and whose results are inserted to complete the artifact.“

Bei der Analyse von MDD-Projekten, deren Werkzeuge auf templatebasierte Generierung zurückgreifen, ist die Betrachtung von Templates elementar. Templates stehen mit anderen Entwicklungsartefakten auf unterschiedliche Art und Weise in Beziehung. Fehlende Werkzeugunterstützung, sowohl bei der Nutzung von Templates als auch bei der Analyse von Templatestrukturen, macht die Betrachtung von Templates mit Hilfe des in dieser Arbeit vorgestellten Ansatzes besonders interessant.

2.3.4. Reports

MontiCore-basierte Generatoren generieren neben dem eigentlichen Zielprodukt Berichte über ihre Ausführung. Diese Reports unterscheiden sich von normalen Log-Dateien dadurch, das relevante Events des Generators und nicht feingranulare Zwischenergebnisse protokolliert werden. Weiterhin werden in den meisten der standardmäßig produzierten Reports Informationen in aggregierter Form bereitgestellt [HR17]. *Werkzeugentwickler* müssen bestimmte Events ähnlich zu dem Aufruf eines Loggers an den Reportingmechanismus von MontiCore melden, damit diese bei Erstellung der Reports mitbetrachtet werden. Andere Events werden hingegen direkt von MontiCore gemeldet.

Listing 2.2 zeigt stellvertretend einen Ausschnitt aus dem Report `01_Summary` [HR17], der für die MontiCore-OD-Sprache der Version 2.2.0 erstellt wurde und einige auf den Reportingevents basierende Statistiken bereitstellt. In der verwendeten Version von MontiCore (4.5.3) werden standardmäßig 22 Reports erzeugt.

```
1 generated files:                    62
2 used standard templates:            76
3 unused standard templates:          26
4 max AST depth:                      9
5 AST node instances:                 240
6 AST node types:                     20
7 variables:                          9
8 variable assignments:               12
```

Listing 2.2: Ausschnitt aus dem Report `01_Summary`, erstellt für die MontiCore OD-Sprache.

Reports dienen dem Zweck, MontiCore-basierte Werkzeuge besser verstehen zu können. Dabei liegen jeweils die Informationen des letzten Ablaufs eines MontiCore-basierten Werkzeugs der Erstellung eines Reports zugrunde. Reports lassen sich somit als Vorstufe der hier vorgestellten Analysen für modellgetriebene Softwareentwicklungsprojekte ansehen. Gleichzeitig lässt sich der Reportingmechanismus aber auch dazu verwenden, die für den vorgestellten Ansatz benötigten Beziehungen zur Generierungszeit zu extrahieren. Diese Möglichkeit ist

durch einen Erweiterungspunkt des Reportingmechanismus, welcher es erlaubt eigene Reports mit einzubinden, gegeben. Die zugehörige Methodik zur Nutzung des Reportingmechanismus für die artefaktbasierte Analyse wird später in Abschnitt 9.2.3 vorgestellt.

2.4. Ausgewählte Modellierungssprachen und modellverarbeitende Werkzeuge

In diesem Abschnitt werden die im Kontext dieser Arbeit relevanten Modellierungssprachen und modellverarbeitenden Werkzeuge vorgestellt. Die Sprachen und Werkzeuge werden als Teil des Ansatzes zur artefaktbasierten Analyse genutzt und/oder sind Analysegegenstand in Anwendungen und Beispielen dieser Arbeit.

2.4.1. UML/P

Die Sprachfamilie UML/P ist ein in [Rum16, Rum17] vorgestelltes Sprachprofil der *Unified Modeling Language* (UML) [OMG15]. Die UML ist ein von der OMG [OMG17b] entwickelter Modellierungsstandard in der Softwareentwicklung. Durch die Vielzahl an Sprachen, deren Anwendungsgebieten und der damit einhergehenden fehlenden Formalisierung eignet sich die UML nicht direkt für die modellgetriebene Softwareentwicklung. Dies konnte jedoch durch die Restriktion von Modellierungssprachen und Sprachkonstrukten in der Sprachfamilie UML/P erreicht werden. Eine textuelle Fassung der UML/P, welche die Nutzung in MDD-Projekten technisch ermöglicht, wurde in [Sch12] entwickelt.

In dieser Arbeit werden zur Realisierung des Ansatzes zur artefaktbasierten Analyse von MDD-Projekten die Sprachen *Class Diagram* (CD), *Object Diagram* (OD) und die *Object Constraint Language* (OCL) verwendet. Des Weiteren enthält die UML/P auch noch die Sprachen *Statechart* (SC), *Sequence Diagram* (SD), eine Sprache zur Definition von Testfällen und die Java/P, eine Java Variante der UML/P [Rum16, Rum17] .

Im Folgenden werden die in dieser Arbeit verwendeten Sprachen kurz vorgestellt. Die Verwendung der Sprachen zur Beschreibung von Artefakten und Artefaktbeziehungen wird später in Abschnitt 5.1 definiert.

Klassendiagramme

Klassendiagramme dienen dazu die Struktur von Softwaresystemen darzustellen. Sie bilden das zentrale Element zur Modellierung von Softwaresystemen mit der UML und sind somit die wichtigste Notation. Mit Hilfe dieser Sprache lassen sich in erster Linie die Klassen und deren Beziehungen modellieren. Darüber hinaus lassen sich auch Enums und Interfaces, zugehörige Eigenschaften wie Attribute, Modifier und Methodensignaturen sowie verschiedene Arten von Beziehungen und deren Kardinaliäten darstellen.

Neben der Verwendung zur Darstellung der technischen, strukturellen Sicht eines Softwaresystems, also der Beschreibung von Quellcodestrukturen, können Klassendiagramme in der Analyse zur Strukturierung von Konzepten der realen Welt genutzt werden [Rum16]. Speziell für diesen Anwendungsfall wurde in [Rot17] eine weitere restriktivere Variante der UML/P Klassendiagramme entwickelt: die Sprache *Class Diagram for Analysis* (CD4A).

In dieser Arbeit wird die CD4A dazu genutzt, Strukturen in modellbasierten Entwicklungsprojekten zu modellieren und somit als Teil des Ansatzes zur artefaktbasierten Analyse verwendet. Darüber hinaus werden in Kapitel 8 graphische UML/P Klassendiagramme dazu genutzt, Implementierungssachverhalte näher zu erläutern.

Objektdiagramme

Objektdiagramme eignen sich für die Spezifikation von exemplarischen Daten eines Softwaresystems. Dabei beschreiben sie einen Zustand des Systems zu einem konkreten Zeitpunkt. Sie können dabei der Struktur eines zugehörigen Klassendiagramms folgen. Eine Prüfung, ob ein Objektdiagramm der vorgegebenen Struktur eines Klassendiagramms entspricht, ist im Allgemeinen nicht trivial. Aus diesem Grund wird in [MRR11a] eine auf Alloy [Jac11] basierende Verifikationstechnik für dieses Problem beschrieben.

In Objektdiagrammen werden Objekte sowie die Verknüpfungen (Links) zwischen Objekten modelliert. Die Modellierung des Objektzustands erfolgt durch Angabe von Attributen und zugewiesenen Werten. Je nach Einsatzzweck können Objektdiagramme eine geforderte, verbotene oder bestehende Situation des Softwaresystems abbilden.

Die aktuelle Version der OD-Sprache (2.2.0) erlaubt zusätzlich zu den in [Sch12] beschriebenen Konzepten die Definition hierarchisch verschachtelter Objekte. Dies hat den Vorteil, dass hierarchische Beziehungen auch als solche in den Objektdiagrammen dargestellt werden können. ODs mit hierarchischen Anteilen werden in dieser Arbeit als *hierarchische* ODs bezeichnet während ODs ohne hierarchische Anteile als *flache* ODs bezeichnet werden. Listing 2.3 zeigt ein Beispiel für ein hierarchisches OD mit zwei verschachtelten Objekten.

Werden Klassendiagramme nicht zur Beschreibung der Klassen einer Implementierung, sondern zur Beschreibung auf konzeptioneller Ebene genutzt, bilden Objekte zugehöriger Objektdiagramme ebenfalls Konzepte der Problemdomäne anstatt Objekte eines Softwaresystems ab. In dieser Arbeit werden Objektdiagramme als Sprache zur Beschreibung von Analysedaten verwendet, d.h., sie bilden den Ist-Zustand des Projekts auf konzeptioneller Ebene ab. Darüber hinaus werden in Abschnitt 7.2.3 Objektdiagramme auch zu Spezifikationszwecken verwendet.

Alle in dieser Arbeit gezeigten Beispiele textueller ODs werden durchgängig in der aktuellen Sprachversion dargestellt. Zu beachten ist, dass einige der in dieser Arbeit vorgestellten Werkzeuge (vgl. Kapitel 8) noch eine ältere Version der OD-Sprache verwenden.

```
objectdiagram Students {                                          OD

  Max:Person {
    id = 1;
    firstName = "Max";
    name = "Mustermann";
    address = address1:Address {
      street = "Ahornstraße";
      city = "Aachen";
      houseNumber = 55;
      country = "Germany";
    };
  };
}
```

Listing 2.3: Beispielhaftes OD mit zwei hierarchisch verschachtelten Objekten.

Object Constraint Language

Die OCL ist eine Spezifikationssprache der UML, mit der sich zusätzliche Bedingungen für Modelle anderer UML Sprachen ausdrücken lassen. So lässt sich die OCL beispielsweise dazu verwenden, Invarianten von Klassendiagrammen, Bedingungen in Sequenzdiagrammen und Vor- bzw. Nachbedingungen von Methoden zu spezifizieren.

Die *OCL Variante der UML/P* (OCL/P) ist eine an Java angelehnte Variante der OCL. In dieser Arbeit wird die OCL/P als einzige Variante der OCL genutzt. Im Rest dieser Arbeit wird OCL als synonym für die OCL/P verwendet. Die Anwendung in dieser Arbeit erfolgt ausschließlich in Kombination mit der Klassendiagrammsprache. OCL-Ausdrücke sind dabei innerhalb von Klassendiagrammartefakten zugelassen.

2.4.2. MontiArc

MontiArc (MA) [HRR12, Hab16] ist eine am Lehrstuhl für Software Engineering der RWTH Aachen entwickelte Architekturbeschreibungssprache. Sie zielt auf die Modellierung von interaktiven verteilten Systemen und von CPS ab. Dabei nutzt sie einen auf FOCUS [BS01] basierenden Architekturstil, welcher selbst zu der Klasse der Komponenten-und-Konnektor (engl. *Component-and-Connector* (C&C)) -Architekturstile zählt. Mittlerweile existieren mehrere Derivate dieser Sprache sowie eine Methodik anhand der sich neue Derivate der Sprache ableiten lassen [BHH+17]. Existierende Ausprägungen der Sprache sind *MontiArcAutomaton* (MAA) [RRRW15], *MontiCar* [KRRvW17], *MontiSecArc* [HHRW15] und *clArc (cloudADL)* [NPR13].

MAA erweitert MA durch Hinzufügen einer Sprache für endliche Automaten (engl. *Finite State Machine* (FSM)) zur Beschreibung von Komponentenverhalten. Insgesamt lassen sich bei

den zu MAA zugehörigen Generatoren folgende Sprachen kombiniert verwenden: Java, CD, Architekturdiagramme (engl. *Architecture Diagram* (ARC)) und FSM. Durch Java oder CDs lassen sich Ports der Architekturdiagramme typisieren. Zugehörige Generatoren ermöglichen die Generierung von ausführbaren Programmen, unter anderem für die Plattformen SmartSoft und ROS [HMSNR+15a].

Ein MAA-basiertes Projekt wird in Abschnitt 4.2 als motivierendes Beispiel eingeführt. Weiterhin stellt das mit MAA realisierte BumperBot-Projekt (siehe Abschnitt 10.2) einen der Anwendungsfälle dieser Arbeit dar.

2.4.3. DEx

DEx [Rot17] ist ein auf MontiCore basierender Generator, der ein Eingabe-CD in ein ausführbares System zur Exploration von Daten transformiert. Die Daten müssen dabei dem durch das Klassendiagramm vorgegeben Schema folgen. Neben der Generierung von Datenhaltungsklassen lässt sich eine graphische Benutzeroberfläche (engl. *Graphical User Interface* (GUI)) und eine Persistenzschicht zur Anbindung der Applikation an eine Datenbank generieren. Dadurch lassen sich Daten sowohl persistent speichern als auch von den Systemnutzern komfortabel explorieren. Die Darstellung der Daten umfasst dabei sowohl die in Abbildung 2.4 dargestellte tabellarische Ansicht als auch eine Formularansicht zum Editieren einzelner Datensätze.

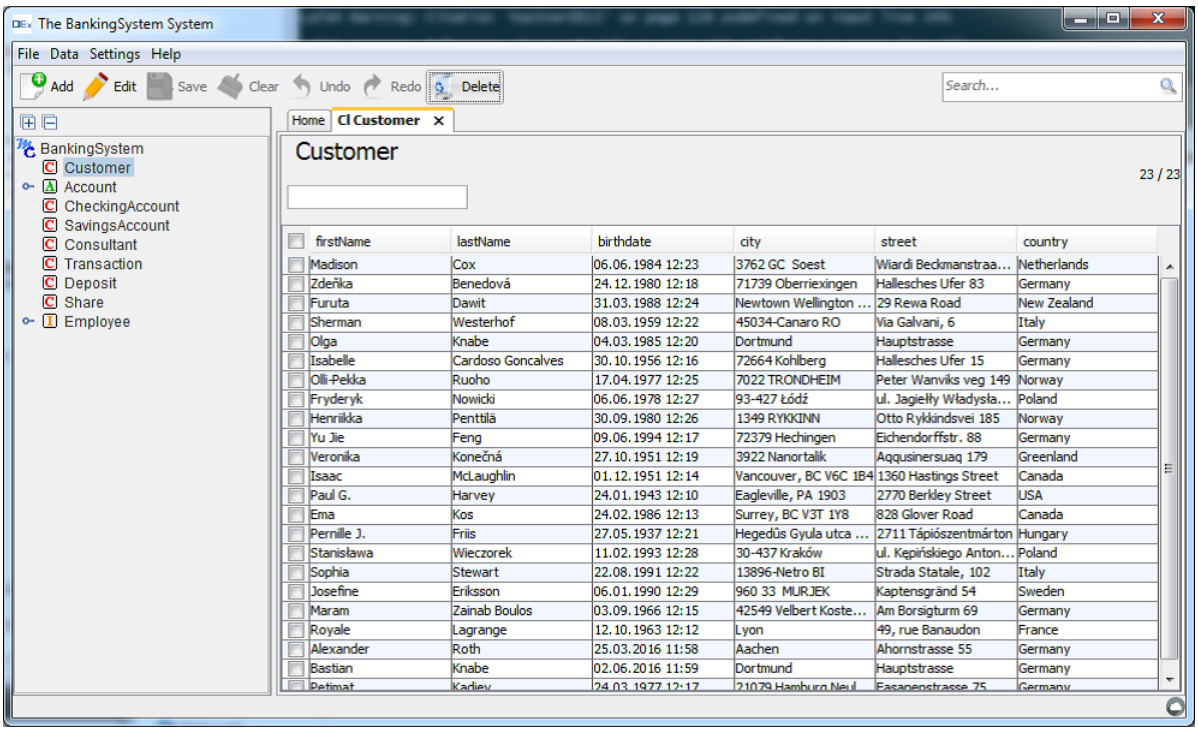

Abbildung 2.4.: Tabellarische Ansicht des DEx-Generators (entnommen aus [Rot17]).

Weiterhin lässt sich per Suchfunktion nach bestimmten Datensätze suchen. DEx verfügt über einen Mechanismus zur Integration von handgeschriebenem Code sowie einen Mechanismus zum Austausch und Hinzufügen von Templates durch den Generatornutzer, wodurch der Generator sehr flexibel und der Generierungsvorgang stark konfigurierbar ist. Weiterhin finden bei Ausführung des Generators interne M2M-Transformationen statt.

Die Konfigurationsfunktionalität und die Durchführung interner Modelltransformationen führen dazu, dass sich Generierungsabläufe in verschiedenen DEx-basierten Entwicklungsprojekten stark unterscheiden können. Aus diesem Grund ist eine Analyse DEx-basierter Projekte besonders interessant. Diese Analyse stellt den in Abschnitt 10.1 beschriebenen Anwendungsfall dar. Weiterhin wird eine DEx-Instanz vorgestellt, die ein Explorieren von in Form von Objektdiagrammen vorliegenden Analysedaten erlaubt (siehe Abschnitt 8.5.1).

Kapitel 3.

Analyse von Softwareentwicklungsprojekten

In diesem Kapitel werden bestehende Techniken zur Analyse von Softwareentwicklungsprojekten vorgestellt. Diese stellen Grundlagen für den entwickelten Ansatz zur artefaktbasierten Analyse dar. Die geforderte Unterstützung von einigen der vorgestellten Techniken wird in den Anforderungen an den Ansatz in Abschnitt 4.5 festgehalten.

3.1. Software Reverse Engineering

Ein Ziel dieser Arbeit ist es, Analysen auf Artefaktstrukturen für bestehende MDD-Projekte zu ermöglichen. Als Voraussetzung für solche Analysen müssen die Beziehungen zwischen den vorhandenen Artefakten expliziert werden. Gerade bei unübersichtlichen Projektsituationen sind die Beziehungen zwischen Artefakten nicht direkt ersichtlich, weshalb Analyseverfahren zur Rekonstruktion der Beziehungen eingesetzt werden. Ein Oberbegriff für solche Analysen bestehender Softwareentwicklungsprojekte ist *Software Reverse Engineering*. In dieser Arbeit wird die folgende Definition für Reverse Engineering aus [CC90] verwendet:

Definition 3.1.1 (Reverse Engineering)**.** *„Reverse engineering is the process of analyzing a subject system to*

- *identify the system's components and their interrelationships and*
- *create representations of the system in another form or at a higher level of abstraction."*

Der Begriff Reverse Engineering ist somit nicht mit *Reengineering* gleichzusetzen. Reenginering bezeichnet den Prozess der Überarbeitung eines bestehenden Softwaresystems oder einzelner Teile davon [BSB08]. Reverse Engineering ist dabei einer der Schritte des Reengineering wie auch die Schritte der Bestandsanalyse, Restrukturierung und Forward Engineering.

In dieser Arbeit wird davon ausgegangen, dass die Quellartefakte des Projekts bei der Analyse frei zugänglich sind. Diese können dazu genutzt werden, um Modelle der vorherrschenden Projektsituation zu erstellen. Andere Anwendungen für Reverse Engineering, die in dieser Arbeit nicht betrachtet werden, sind die Rückgewinnung von Quellcode aus einem bereits in

kompilierter Form vorliegenden Programm sowie die Erschließung von Informationen wie beispielsweise des Kommunikationsprotokolls bei der Beobachtung von Kommunikationskanälen.

Darüber hinaus ist der Kontext der Arbeit die Analyse von MDD-Projekten mit der in Abschnitt 2.2 definierten Bedeutung. Dies bedeutet, dass nicht nur die Quellen des Zielprodukts, sondern auch die Quellartefakte der beteiligten Werkzeuge und die Zusammenhänge zwischen den Artefakten dieser Systeme Analysegegenstand der in Kapitel 7 beschriebenen Analysemethodik sind.

3.2. Traceability und Artefaktabhängigkeiten

Traceability, *Tracing* oder *Tracelink* sind Begriffe, die Beziehungen zwischen Artefakten des Softwareentwicklungsprozesses beschreiben oder mit diesen in Zusammenhang stehen. Im Folgenden werden verschiedenen Definitionen der Begriffe diskutiert und in den Kontext dieser Arbeit eingeordnet.

Während in [Bro18] ein Trace als eine Sequenz von gerichteten Links definiert ist, bei dem das Ziel eines Links der Sequenz mit der Quelle des darauffolgenden Links übereinstimmt, wird in den nachfolgend betrachteten Definitionen aus [CHGZ12] ein einzelner gerichteter Link bereits als Tracelink (kurz Trace) bezeichnet.

- **Traceability.** *„The potential for traces to be established and used. Traceability (i.e., trace "ability") is thereby an attribute of an artifact or of a collection of artifacts. Where there is traceability, tracing can be undertaken and the specified artifacts should be traceable."*
- **Tracing.** *„The activity of either establishing or using traces."*
- **Vertical tracing.** *„In software and systems engineering contexts, the term is commonly used when tracing artifacts at differing levels of abstraction so as to accommodate life cycle-wide or end-to-end traceability, such as from requirements to code. "*
- **Horizontal tracing.** *In software and systems engineering contexts, the term is commonly used when tracing artifacts at the same level of abstraction [. . .]"*

Diese Definitionen trennen zwar zwischen der Fähigkeit des Tracings (Traceability) und der Erstellung bzw. Nutzung des Tracings, unterscheiden jedoch nicht zwischen verschiedenen Linkarten. Stattdessen lassen sich grundsätzlich alle Beziehungen zwischen Entwicklungsartefakten als Traces bezeichnen. Es wird lediglich zwischen zwei Unterkategorien (horizontal und vertikal) von Tracings unterschieden und nicht klar zwischen dem Erstellen und der Nutzung von Traces getrennt.

In [SI16] wird hingegen zwischen fünf verschiedenen Bereichen für vertikales Tracing unterschieden: Zwischen Anforderungen und Quellcode, zwischen Anforderungen und Design, zwischen Anforderungen und Testfällen, zwischen Anforderungen und Fehlerberichten und

zwischen Design und Quellcode. Durch solche Klassifizierungen lassen sich genauere Aussagen und Analysen über die Situation des MDD-Projekts treffen. Die Typisierung von Artefakt- und Beziehungsarten ist ein wichtiger Bestandteil des entwickelten Ansatzes und Voraussetzung, um bestimmte Analysen überhaupt durchführen zu können.

Neben der Verwendung des Tracebegriffs für statische Strukturen, findet der Begriff auch bei der Ausführung von Softwaresystemen Verwendung. Während bei statischen Strukturen ein Trace jegliche Art von Zusammenhang zwischen Artefakten darstellen kann, werden bei Laufzeitbetrachtungen Sequenzen der Programmausführung als Programmtraces bezeichnet. So ist nach [KWK10] Tracing (*Trace Recording*) wie folgt definiert: *„[. . .] trace recording implies detection and storage of relevant events during run-time, for later off-line analysis.“* Somit wird bei Verwendung dieser Definition die Menge an Beziehungen eingeschränkt. In dieser Arbeit sind vor allem die Beziehungen zwischen Artefakten von Interesse. Diese lassen sich auch als Artefakttraces verstehen, die nicht durch Programmabläufe des Softwareprodukts, sondern durch voranschreitende Buildprozesse induziert werden. In Abschnitt 6.5.1 werden aber auch Aktionen und Events zur Generierungszeit betrachtet, mit deren Hilfe sich entsprechende Artefaktbeziehungen in MDD-Projekten ableiten lassen.

Im *Requirements Engineering* ist Traceability nach [DHJ05] wie folgt definiert: *„In the requirements engineering context, traceability is about understanding how high-level requirements – objectives, goals, aims, aspirations, expectations, needs – are transformed into low-level requirements.“* Dabei wird zwischen vier verschiedenen Ebenen von Anforderungen unterschieden: *Stakeholderanforderungen*, *Systemanforderungen*, *Subsystemanforderungen* und *Komponentenanforderungen*. Weiterhin können Testfälle zu den Anforderungen auf allen vier Ebenen durch einen Trace verknüpft werden. In der Automobilindustrie wird Tracing im Entwicklungsprozess zwischen Entwicklungsartefakten zum Beispiel durch Automotive SPICE [MHDZ16] gefordert. Automotive SPICE fordert dabei sowohl vertikales Tracing (z.B. zwischen Softwareanforderungen und -architektur) als auch horizontales Tracing (z.B. zwischen Softwaremodulen und Testspezifikationen). Ein stellvertretendes Werkzeug welches das Tracing von Artefakten im Entwicklungsprozess erlaubt ist *YAKINDU Traceability* [Yak17a]. Dort ist es dem Anwender überlassen, Artefaktarten und Tracearten zu spezifizieren und zugehörige Daten in dem Werkzeug zu verwalten. Ein ähnlicher Ansatz wird auch in dieser Arbeit verfolgt. Zusätzlich wurde ein Modell für Artefakt- und Beziehungsarten in MDD-Projekten sowie eine Werkzeugkette zur automatischen Analyse von MDD-Projekten entwickelt.

Der Unterschied zwischen Tracing in MDD-Projekten und Tracing in anderen Projekten ist durch den hohen Grad der Automatisierung und der Existenz zusätzlicher Artefaktarten in MDD-Projekten bestimmt. Während in anderen Projekten die Erstellung von Artefakten oftmals manuell vorgenommen wird, werden in MDD-Projekten Artefakte durch den Einsatz von M2M- und M2T-Transformationen zu einem erheblichen Teil automatisch erstellt, zusammengeführt oder aufgeteilt. Dafür existieren dort, bedingt durch die zusätzlichen Artefaktarten, weitere Tracearten, die ebenfalls bei einer Analyse mit betrachtet werden müssen. In anderen Projekten müssen Traces per Hand nach Erstellung der Artefakte hinzugefügt und gewartet werden, während bei dem Einsatz von Transformationen Traces direkt mit erstellt werden können.

Gerade wenn mit wiederholbaren Transformationen (im Gegensatz zur einmaligen Generierung und manuellen Weiterverarbeitung) gearbeitet wird, ist dies zu empfehlen. Ansonsten muss bei jeder neuen Ausführung des Buildprozesses der Trace erneut manuell hinzugefügt oder seine Korrektheit zumindest überprüft werden.

In [GS17] wird Tracing zwischen an Transformationen beteiligten Artefakten als *Key Challenge* im Bereich der Erstellung und Wartung von Modelltransformationen bezeichnet. Um Werkzeuge für MDD-Projekte (weiter-) entwickeln zu können, ist ein Verständnis der Beziehungen zwischen den beteiligten Artefakten essentiell. Die Kenntnis von Tracingbeziehungen ermöglicht unter anderem den Aufwand von Metamodell- und Transformationsevolutionen abzuschätzen, Transformationen zu debuggen und Refactoringpotenzial in Transformationen zu finden [GS17]. Diese Anwendungsdomäne stimmt mit der Anwendungsdomäne in dieser Arbeit überein. Somit kann diese Arbeit als Schritt zur Lösung der *Key Challenge* angesehen werden.

Der Begriff der Artefaktabhängigkeit ist genau wie der Tracebegriff je nach Kontext anders und somit nicht eindeutig definiert. In dieser Arbeit wird die folgende Definition verwendet:

Definition 3.2.1 (Artefaktabhängigkeit)**.** *Zwei Artefakte A und B hängen genau dann voneinander ab, wenn sie in einer der folgenden Beziehungen zueinander stehen:*

- *A hat statische Kenntnis von B und nutzt ggf. von B exportierte Symbole (A bezieht sich auf B).*
- *A wird von einem Werkzeug genutzt, um B zu produzieren (A produziert B).*
- *A ist ein Artefakt, das selbst aus weiteren Artefakten besteht und B enthält (A enthält B).*

Ein Artefakt A, dass sich auf ein anderes Artefakt B bezieht, nutzt dessen Existenz oder dessen Inhalt, um seinen eigenen Zweck zu erfüllen. Bei einer Nutzung des Inhalts ist es erforderlich, dass die von A genutzten Symbole von B entsprechend angeboten (exportiert) werden. Ein Beispiel hierfür ist das Importieren von Java-Dateien bei dem die Symbole der importierten Datei entsprechend genutzt werden können. Ein Artefakt produziert ein anderes Artefakt, wenn es als Eingabe für ein Werkzeug dient, das während seiner Ausführung das andere Artefakt erstellt. Bei der dritten Art von Artefaktabhängigkeit beinhalten Artefakte andere Artefakte. Beispiele für Artefakte, die andere Artefakte beinhalten, stellen Archive dar.

In Kapitel 5 werden diese drei Beziehungsarten genauer betrachtet. Sie können abhängig von den beteiligten Artefakten spezifische Ausprägungen haben. Dabei ließen sich alle bisher gefundenen und betrachteten Beziehungsarten in eine der drei vorgestellten Kategorien einteilen. In Kapitel 6 werden spezifische Ausprägungen dieser Beziehungen im Kontext modellgetriebener Entwicklungsprojekte zusammen mit den beteiligten Artefaktarten vorgestellt.

Tracelinks stellen Artefaktabhängigkeiten explizit dar. Vor ihrer Nutzung (siehe Abschnitt 3.4) müssen sie zunächst extrahiert werden (siehe Abschnitt 3.3). Beziehungen zwischen Artefakten können aber auch bestehen, wenn diese nicht durch Tracelinks explizit sichtbar gemacht wurden.

Da sowohl der Tracebegriff als auch der Abhängigkeitsbegriff Obermengen von verschiedener Beziehungsarten zwischen Artefakten repräsentieren, deren Unterarten sich jedoch fundamental voneinander unterscheiden können, ist die Verwendung dieser Begriffe an den meisten Stellen dieser Arbeit zu unspezifisch und unpräzise. An den Stellen, an denen dennoch von einem der Begriffe ohne weitere Erläuterungen Gebrauch gemacht wird, ist jeweils die Obermenge der Artefaktbeziehungen nach Definition 3.2.1 gemeint.

3.3. Extraktion von Traceabilityinformationen

Traces können entweder explizit angegeben oder in manchen Fällen auch automatisch erhoben werden. Eine solche Extraktion ist Grundlage oder Teil bestimmter Analysearten. Nach [BSB08] gibt es folgende Arten der Softwareanalyse: die statische Analyse, die dynamische Analyse und die evolutionäre Analyse. Während die statische Analyse und die dynamische Analyse eine Extraktion von Traceabilityinformationen beinhalten, ist dies für die evolutionäre Analyse nicht zwingend der Fall. Diese betrachtet stattdessen die Historie der Software und vergleicht verschiedene Versionsstände miteinander. Die evolutionäre Analyse kann dabei helfen den Zeitpunkt und Grund für ungewollte Änderungen im Nachhinein zu rekonstruieren. Diese Technik lässt sich gut im Zusammenhang mit Metriken (siehe Abschnitt 3.6) nutzen, da Metriken Zahlenwerte berechnen, wodurch ein Vergleich zwischen zwei Zeitpunkten einfach ist. Es ist jedoch auch möglich die Tracestrukturen miteinander zu vergleichen. Die in dieser Arbeit entwickelte Analysemethodik sieht eine evolutionäre Analyse nicht vor. Dies ist durch die Wahl der Modellierungstechnik (siehe Kapitel 5) begründet. Diese unterstützt nur eine Abbildung der aktuellen Projektsituation. Aufbauend auf dieser Arbeit könnte aber eine Erweiterung der Modellierungstechnik untersucht werden (siehe Abschnitt 11.2), um evolutionäre Analysen auf Artefaktstrukturen von MDD-Projekten durchführen zu können. Im Folgenden werden die beiden Verfahren zur Extraktion von Traces vorgestellt. Da in dieser Arbeit zwischen der Extraktion von Traces und deren Analyse bzw. Nutzung getrennt wird, werden für die automatische Erhebung von Traces die Begriffe statische und dynamische Extraktion verwendet. In [TNL17] wird eine Studie vorgestellt in der beide Extraktionsansätze zum Zweck einer Architekturkonformitätsprüfung (engl. *Architecture Consistency Checking* (ACC); siehe auch Abschnitt 3.5) systematisch verglichen werden. Die Studie kommt zu dem Schluss, dass die statische Variante weniger aufwändig und wartungsintensiv ist und damit bereits einen Großteil der Architekturverletzungen identifiziert werden können. Trotzdem reichen statische Analysen nicht aus, da weder Interprozesskommunikationen mit in Betracht gezogen werden noch die Häufigkeit von Architekturverletzungen bestimmt werden kann. Somit haben beide Ansätze ihre Daseinsberechtigung.

3.3.1. Statische Extraktion

Bei der statischen Extraktion werden die relevanten Artefakte statisch ohne Ausführung der Artefakte analysiert. Dabei können sowohl Eigenschaften des jeweiligen Artefakts als auch Beziehungen zwischen Artefakten extrahiert werden. Der Zweck, für den eine statische Analyse durchgeführt wird, kann sehr unterschiedlich sein. SonarQube [CP13] führt beispielsweise statische Analysen durch, um die Testabdeckung zu bestimmen, die Einhaltung von Codierregeln (Coding Conventions) zu prüfen sowie um Quelltextklone und Code Smells aufzuspüren. Darüber hinaus lassen sich durch Reverse-Engineering-Techniken Modelle aus dem Quellcode extrahieren. Integrierte Entwicklungsumgebungen (engl. *Integrated Development Environment* (IDE)) wie Eclipse [Ecl17], IntelliJ [Int17a] oder Visual Studio [Vis17] unterstützen beispielsweise die automatische Erstellung von UML-Klassendiagrammen aus Quellcode. Eine vollautomatische Erstellung von Architekturmodellen ist hingegen schwieriger, da dort auf die von einem Menschen konzipierte Softwarearchitektur geschlossen werden muss. In [GIM13] werden entsprechende Ansätze in einer Studie verglichen.

Ein Beispiel für ein Werkzeug, das eine statische Extraktion durchführt, wird ab Java 8 zusammen mit dem JDK ausgeliefert. Das Kommandozeilenwerkzeug jdeps analysiert *.class* Dateien und berechnet die Abhängigkeiten zwischen Paketen, Klassen und Jar-Dateien. Die Ausgabe wird dabei standardmäßig auf die Konsole geschrieben, lässt sich jedoch auch im DOT-Format ausgeben [Sha14]. Dadurch lässt sich die Ausgabe z.B. durch das Visualisierungswerkzeug Graphviz [Gra17] graphisch darstellen.

3.3.2. Dynamische Extraktion

Im Gegensatz zur statischen Analyse werden die Informationen bei der dynamischen Extraktion durch Beobachtungen bei der Ausführung der Artefakte gewonnen. So lassen sich beispielsweise Modellsimulationen oder ausführbare Softwaresysteme beobachten, um Aussagen über Abläufe bzw. Sequenzen der Ausführung machen zu können. Es lassen sich auch Ressourcen- und Lastanalysen durch dynamische Analysen durchführen. Diese Art der Analyse lässt sich aber auch für die Fehlersuche und zur Beobachtung von Interaktionen zwischen Systemkomponenten verwenden [NLG+15].

Für die dynamische Analyse kann eine Instrumentierung der Software nötig sein, um auf die erforderlichen Informationen zur Laufzeit zugreifen zu können. Mit Instrumentierung wird die Anreicherung der bestehenden Software um zusätzlichen Programmcode zu Analysezwecken bezeichnet. Wird bei Ausführung der Software bereits detailliertes Logging betrieben, kann auch durch Analyse der Logdateien das Laufzeitverhalten rekonstruiert werden. Darüber hinaus gibt es Ansätze wie Kieker [vHWH12, vHRH+09], ein Framework für das Beobachten von Softwarelaufzeitverhalten, bei denen keine Instrumentierung notwendig ist. Zu beachten ist jedoch, dass auch die Verwendung solcher Ansätze geringe Auswirkungen auf die Performance des zu untersuchenden Systems hat. Der in dieser Arbeit beschriebene Ansatz sieht eine Instrumentierung des zu untersuchenden Systems vor. Durch Verwendung des Reportingme-

chanismus (siehe Abschnitt 2.3.4) kann jedoch auf die bereits vorhandene und vorgesehene Instrumentierung der Werkzeuge zurückgegriffen werden, sodass die Lesbarkeit des Quellcodes nicht zusätzlich beeinträchtigt wird.

In dieser Arbeit werden Artefaktdaten sowohl statisch als auch dynamisch extrahiert. Diese lassen sich in Kombination betrachten, wodurch kombinierte Analysen der Daten aus beiden Quellen möglichen werden.

3.4. Nutzung von Traceabilityinformationen

Nach der Bereitstellung von Traceabilityinformationen können diese für vielfältige Analysen genutzt werden. In diesem Abschnitt werden einige bekannte Analysen aus dem Bereich Software Engineering vorgestellt.

3.4.1. Abhängigkeitsanalyse

Für den Begriff Abhängigkeitsanalyse gibt es mehrere Bedeutungen. Eine mögliche Bedeutung bezeichnet das Extrahieren von Artefakt-, Modul- oder Systembeziehungen, wodurch diese Beziehungen explizit gemacht werden [CAvdSA11]. Eine solche Extraktion kann sowohl statische als dynamische durchgeführt werden. Diese Bedeutung des Begriffs Abhängigkeitsanalyse wird in dieser Arbeit nicht weiter verwendet. Stattdessen wird für die Erhebung von Beziehungen (oder auch Extraktion von Traceabilityinformationen, vgl. Abschnitt 3.3) der Begriff Extraktion verwendet.

In einer anderen Bedeutung wird mit dem Begriff Abhängigkeitsanalyse die Analyse aller transitiv abhängigen Artefakte eines untersuchten Artefakts bezeichnet [IBM17]. Die Motivation hinter einer solchen Analyse ist die Wiederverwendung des Artefakts in einem anderen Kontext. Abhängige Artefakte müssen in dem neuen Anwendungskontext ebenfalls zur Verfügung gestellt werden. Um die Analyse basierend auf Traceabilityinformationen durchzuführen, muss eine rekursive Vorwärtssuche entlang der Traces erfolgen. Eine solche Analyse kann sowohl auf allen Abhängigkeitsarten oder auf einer der beiden Beziehungskategorien (vgl. Abschnitt 3.2) durchgeführt werden. Auch ist es denkbar, dass eine solche Analyse nur auf spezifischen Ausprägungen (siehe Kapitel 5) dieser abstrakteren Beziehungsarten durchgeführt wird. In Abschnitt 7.2.2 wird eine solche Analyse für alle *bezieht-sich-auf*-Beziehungen (*refersTo*) unter Verwendung der OCL spezifiziert.

3.4.2. Change-Impact-Analyse

Wenn bestehende Software verändert werden muss, müssen vor der Implementierung der Änderungen zunächst die Auswirkungen der Änderungen analysiert werden. Dies geschieht üblicherweise mit einer Änderungsauswirkungsanalyse (engl. *Change-Impact-Analysis*). Der Begriff wurde in [BA96] geprägt und wurde in [IEE05] wie folgt definiert:

„impact analysis: Identifies all system and software products that a change request affects and develops an estimate of the resources needed to accomplish the change. This includes determining the scope of the changes to plan and implement work, accurately estimating the resources needed to perform the work, and analyzing the requested changes' cost and benefits.“

Change-Impact-Analysen können wie oben beschrieben im Kontext des gesamten Entwicklungsprozess durchgeführt werden, um vorab festzustellen, welche Auswirkungen die Änderung eines bestimmten Artefakts nach sich ziehen. Ein Beispiel hierfür ist das Analysieren von notwendigen Änderungen an weiteren Artefakten bei Hinzufügen oder Ändern einer einzelnen Anforderung.

Im Kontext dieser Arbeit ist die Durchführung einer Change-Impact-Analyse auf Artefakten des MDD-Projekts von Interesse. Zur Durchführung der Analyse lässt sich auf vorliegenden Traceabilityinformationen eine Rückwärtssuche durchführen, sodass alle transitiv verknüpften Artefakte berechnet werden. Wie in Abschnitt 2.2 beschrieben, wird jedoch nicht auf die Abschätzung des Aufwands, der Kosten und des Nutzens fokussiert. In Abschnitt 7.2.2 wird eine solche Analyse für alle *bezieht-sich-auf*-Beziehungen (vgl. Abschnitt 3.2) unter Verwendung der OCL spezifiziert.

3.4.3. Abdeckungsanalyse

Mit Hilfe der Abdeckungsanalyse [MID17] wird ermittelt, ob der Informationsgehalt eines Artefakts durch ein oder mehrere andere Artefakte abgedeckt ist. Für spezielle Artefaktformen kann durch den Entwicklungsprozess eine solche Abdeckung verlangt werden. Ein Beispiel hierfür ist, dass Anforderungen auf einer höheren Ebene durch Anforderungen der darunterliegenden Ebene abgedeckt sein müssen. Auch sollte jede Anforderung durch einen Testfall abgedeckt sein. Es kann aber auch notwendig sein mehrere Traces über mehrere Ebenen hinweg nachzuverfolgen, um eine Abdeckung zu bestätigen oder das Fehlen einer Abdeckung offenzulegen.

Bei der Durchführung einer solchen Analyse müssen spezielle Arten von Traces beachtet werden. Dabei wird, wie bei den anderen in diesem Abschnitt vorgestellten Analysearten, von einem Quellartefakt ausgegangen. Normalerweise geht diese Art von Traces vom abdeckenden Artefakt zum abgedeckten Artefakt, da das abdeckende Artefakt vom abgedeckten Artefakt abhängt. Somit ist in diesem Fall eine Rückwärtssuche unter Betrachtung ausgewählter Tracearten notwendig. Prinzipiell lässt sich diese Art der Analyse auch mit der entwickelten Methodik durchführen. In dem in Abschnitt 4.2 vorgestellten Szenario wird eine solche Abdeckungsanalyse betrachtet.

3.5. Architekturrekonstruktion und Architekturkonformität

Architekturrekonstruktion und die Prüfung der Architekturkonformität sind Techniken, die im Bereich der Softwarearchitektur angewandt werden, um den aktuellen Status der Architektur (die *Ist-Architektur* oder *Descriptive Architecture*) zu erheben bzw. diese mit der vorgesehenen Architektur (auch *Soll-Architektur* oder *Prescriptive Arcitecture*) zu vergleichen. Die folgenden Definitionen grundlegender Begriffe sind [TMD09] entnommen.

Definition 3.5.1 (Architekturdegradation)**.** *„The resulting discrepancy between a system's prescriptive and descriptive architecture is referred to as* architectural degradation*."*

Definition 3.5.2 (Architekturdrift)**.** *„*Architectural Drift *is introduction of principal design decisions into a systems's descriptive architecture that (a) are not included in, encompassed by, or implied by the prescriptive architecture, but which do not violate any of the prescriptive architecture's design decisions."*

Definition 3.5.3 (Architekturerosion)**.** *„*Architectural erosion *is the introduction of architectural design decisions into a system's descriptive architecture that violate its prescriptive architecture."*

In dieser Arbeit wird das Ergebnis der Berechnung der Architekturdegradation als *Differenzarchitektur* bezeichnet. Eine solche Differenzarchitektur enthält somit alle Unterschiede zwischen der Soll- und der Ist-Architektur und vereinigt die Anteile des Architekturdrifts mit denen der Architekturerosion. Dabei wird in dieser Arbeit das Fehlen von vorhergesehenen Beziehungen als Architekturdrift, zusätzliche unvorhergesehene Beziehungen als Architekturerosion angesehen. Um an die Differenzarchitektur zu gelangen, muss die Ist-Architektur mit der Soll-Architektur vergleichen werden. Die Soll-Architektur muss zu diesem Zweck manuell spezifiziert werden. Demgegenüber existieren Techniken, die eine (semi-) automatische Rekonstruktion der Ist-Architektur aus dem Quellcode erlauben. Dieser Vorgang wird als *Architekturrekonstruktion* bezeichnet. Bei der Architekturrekonstruktion unterscheidet man zwischen Top-down- und Bottom-up-Ansätzen. Bottom-up-Ansätze werden auch als *architecture recovery* bezeichnet [DP09]. Sie arbeiten vollautomatisch und können daher auch dann eingesetzt werden, wenn keine Soll-Architektur vorhanden ist, leiden dafür aber an fehlender Genauigkeit [GIM13]. In [vDB11, PvDB12] wurden zwei Bottom-up-Ansätze kombiniert, um die Qualität der Ist-Architektur während der Architekturrekonstruktion zu verbessern, wodurch auch die Güte der rekonstruierten Architektur gesteigert werden konnte. Im Gegensatz zu Bottom-up-Ansätzen wird bei Top-Down-Ansätzen die Existenz einer modellierten Soll-Architektur vorausgesetzt. Top-down-Ansätze werden auch als *architecture discovery* bezeichnet [DP09].

Die Berechnung der Differenzarchitektur ausgehend von einer Soll-Architektur und dem Programmquellcode wird auch als Prüfung der Architekturkonformität (engl.: *Architecture*

Consistency Checking (ACC)) bezeichnet [PKB13]. In einer aktuellen Studie [ABO+17] werden u.A. verschiedene ACC-Ansätze aus der Forschung verglichen. Dabei wurden die Ansätze anhand folgender Merkmale klassifiziert:

- Art des zugrundeliegenden Extraktionsverfahrens (statisch, dynamisch). Alle Ansätze der Architekturrekonstruktion nutzen Extraktionsverfahren, um die Beziehungen zwischen den atomaren Architektureinheiten (z.B. Typen, Dateien) zu erheben. Es existiert mit [SSC96] auch ein Ansatz der beide Extraktionsverfahren kombiniert. Dort werden Informationen aus einer dynamischen Extraktion dazu genutzt die Häufigkeit und somit Relevanz von Architekturverletzungen, die mit Hilfe einer statische Extraktion identifiziert wurden, zu bewerten.
- Technik zur Bewertung der Architekturkonsistenz. Besonders interessante Techniken sind dabei:
  - *Reflexion Modeling* (RM) [MNS01], eine Technik bei der vorliegende Quellcodeartefakte Architektureinheiten der Soll-Architektur zugeordnet werden. Aus den Beziehungen zwischen den Quellcodeartefakten wird dann auf die Beziehungen der Ist-Architektur geschlossen.
  - DSL-basierte Ansätze wie [HZ12, GMR15], ermöglichen eine komfortable Definition der Soll-Architektur. Neben der Angabe von Architektureinheiten und erwarteten Beziehungen lassen sich zusätzlich auch Informationen wie komplexe Bedingungen für Architekturverletzungen oder die Schwere von Architekturverletzungen im Falle eines Auftretens spezifizieren. DSL-basierte Ansätze lassen sich mit RM-basierten Ansätzen kombinieren.

Diese Ansätze lassen sich auch für Architekturen außerhalb des Zielprodukts einsetzen. Dazu muss der Umfang der betrachteten Beziehungen ausgeweitet werden. In MDD-Projekten ist neben der Architektur des Zielprodukts, auch die Architektur einzelner MDD-Werkzeuge, das Zusammenspiel zwischen Werkzeugen und Artefakten sowie die Zusammensetzung der Werkzeuge zu Werkzeugketten für solche Analysen interessant.

In dieser Arbeit wird ein flexibler, RM-basierter Ansatz zur Prüfung der Architekturkonformität für MDD-Projekte vorgestellt (siehe. Abschnitt 7.2.3). Dabei können je nach Anwendungsgebiet statische und/oder (auch in Kombination) dynamische extrahierte Beziehungsarten verwendet werden.

3.6. Metriken in der Softwareentwicklung

Metriken werden in der Softwareentwicklung vor allem dazu genutzt, Maße für die Softwarequalität zu berechnen. Metriken bilden somit bestimmte Eigenschaften der Software auf einen Zahlenwert ab. Das Ziel ist dabei, die Softwarequalität zu prüfen und gegebenenfalls Verbesserungsmaßnahmen einzuleiten. Im Folgenden wir die Definition nach [IEE98] verwendet:

Definition 3.6.1 (Metrik). *„A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.“*

Neben der oben genannten Anwendung von Metriken auf den Quellcode werden Metriken in der Softwareentwicklung auch für die Bewertung des Entwicklungsprozesses oder die Erhebung vor Informationen für die Projektsteuerung eingesetzt. [Lig09] unterteilt die Metriken (Maße) daher in die folgenden Kategorien: *Prozessmaße*, *Produktmaße* und *Projektmaße*.

Produktmetriken quantifizieren üblicherweise die Eigenschaften des zu entwickeln Softwaresystems, während Prozessmetriken den Entwicklungsprozess auf Eigenschaften wie Produktivität, Kostenverteilung oder Dauer von Arbeitsschritten hin untersuchen [BSB08]. In dieser Arbeit werden Metriken dazu verwendet, Aussagen über den Buildprozess in MDD-Projekten zu machen. Dabei schließt die Anwendung auf den Buildprozess auch das Zielprodukt mit ein. Dies ist sinnvoll, da der Buildprozess automatisch durch MDD-Werkzeuge durchgeführt wird. Diese können ebenfalls agil weiterentwickelt werden, was wiederum Auswirkungen auf das Zielprodukt haben kann. Somit kommen in diesem Zusammenhang Prozess- und Produktmetriken zum Einsatz.

Der in dieser Arbeit entwickelte Ansatz ermöglicht es dem Anwender Metriken zu spezifizieren, diese automatisch auswerten zu lassen und die Ergebnisse tabellarisch und graphisch darzustellen. Zur Entwicklung neuer Metriken kann beispielsweise die Methodik aus [vSBCR02] verwendet werden.

In Tabelle 3.1 werden Beispiele bekannter Vertreter aus der Klasse der *Produktmetriken* dargestellt. Diese lassen sich nach [BSB08] weiter in Größen-/Volumenmetriken und Struktur-/Komplexitätsmetriken unterteilen. Zusätzlich wurde die Testüberdeckung als eigene Metrikart hinzugefügt. Details zu den aufgeführten Vertretern dieser Metrikart lassen sich beispielsweise [Hof13] entnehmen.

Metriken lassen sich nicht nur für einzelne Artefakte, sondern auch für Architektureinheiten des Softwaresystems berechnen. Bei der Berechnung solcher Metriken ist zu beachten, dass Module nicht zwingend automatisch aus dem Quellcode hervorgehen (siehe Abschnitt 3.5). Grundsätzlich gilt, dass eine detailliertere Unterscheidung zwischen Artefaktarten eine Spezifikation von konkreteren Metriken begünstigt. Im Fall von objektorientierten Programmiersprachen kann beispielsweise zusätzlich zu der Anzahl von Codezeilen auch die Anzahl der Attribute, die Tiefe der Vererbungshierarchie oder die Anzahl überschriebener Methoden berechnet werden [BSB08].

[1] Der Begriff technische Schuld wurde durch [Cun92] geprägt. SonarQube [CP13] berechnet die technische Schuld als Aufwand (Zeit), welcher nötig ist, um alle offenen, erkannten Punkte bezüglich der Wartbarkeit zu beheben. Jeder Art von Problem wird dabei eine eigene technische Schuld zugewiesen, sodass zur Berechnung der Gesamtschuld lediglich die Summe der Aufwände berechnet werden muss. (siehe `https://docs.sonarqube.org/display/SONAR/Metric+Definitions`)

Metrikart	Metrik
Größen-/Volumenmetriken	Anzahl Zeilen pro Modul
	Anzahl Zeilen pro Artefakt
	Anzahl Zeilen pro Methode
	Anzahl Artefakte pro Modul
	...
Struktur-/Komplexitätsmetriken	Halstead [Hal77]
	Zyklomatische Komplexität (McCabe) [McC76]
	Kohäsion und Kopplung [Lar04]
	Technische Schuld [1]
	...
Testüberdeckung [Hof13]	Anweisungsüberdeckung
	Zweigüberdeckung
	Pfadüberdeckung
	Bedingungsüberdeckung
	...

Tabelle 3.1.: Beispiele bekannter Metriken aus der Klasse der Produkt- und Testüberdeckungsmetriken.

3.7. Visualisierung von Abhängigkeiten in der Softwareentwicklung

Es existieren verschiedene Visualisierungstechniken zur Darstellung von Abhängigkeiten (vgl. Abschnitt 3.2) in der Softwareentwicklung. Dabei können sowohl die extrahierten Traces als auch berechnete Architekturmodule und deren Verbindungen dargestellt werden. Einen Ansatz, wie sich diese Darstellungsarten zu einem ganzheitlichen Visualisierungswerkzeug für MDD-Projekte integrieren lassen, wurde in [GLR17] beschrieben. Die Möglichkeit zur graphischen Darstellung von UML-Diagrammen, die auch für diesen Zweck genutzt werden kann, wird als bekannt vorausgesetzt und nicht weiter vorgestellt.

3.7.1. Abhängigkeitsmatrix

Die Abhängigkeitsmatrix stellt Beziehungen zwischen einer Menge von Artefakten oder Modulen in einer Matrix dar. Dabei repräsentieren die Zeilen und Spalten der Matrix mit dem gleichen Index jeweils dasselbe Artefakt. Ein Beispiel einer solchen Matrix aus dem Werkzeug *Structure101* [Str17a] wird in Abbildung 3.2 dargestellt.

Die Anzahl an Beziehungen von einem Element in einer Spalte zu einem Element in einer Zeile wird in die jeweilige Zelle eingetragen. Somit ist es möglich die Beziehungen zwischen den dargestellten Elementen auf einen Blick festzustellen. Diese Darstellungsart eignet sich besonders gut für große Datenmengen, bei denen die Darstellung als Diagramm zu unübersichtlich ist.

	.views.freemarker	.impl	.views.jsp.ui	.views.freemarker...	.spi	.views.jsp	.components.table	.components	.components.tabl...	.dispatcher.mapper	.components.tem...	.portlet.util	.dispatcher	.portlet.context	.views.velocity	.dispatcher.multipart	.util	.views.util	.config	.struts2
.views.freemarker											3		2							
.impl													4							
.views.jsp.ui																	4	1		
.views.freemarker.tags	2																			
.spi		26																		
.views.jsp			35					13												
.components.table				2					10											
.components			...	...		92	8				5									
.components.table.r...							10													
.dispatcher.mapper						3		8					16							
.components.template								7												
.portlet.util								3												
.dispatcher	4	2				10		16		6						2	6			4
.portlet.context	9							3				7								
.views.velocity											4		5				2			
.dispatcher.multipart													3							
.util						5		36		6			6		3	1		2		
.views.util	3					1		7					1		1		2			
.config	5							6		4	2		19		9	1		4		3
.struts2	13	24			1	1	2	13		4			11	1	5			1	4	

Abbildung 3.2.: Darstellung von Abhängigkeiten mit Hilfe einer Abhängigkeitsmatrix (entnommen von [Str17b]).

Auch ist diese Betrachtungsart besonders für die Darstellung einer Schichtarchitektur geeignet, bei der eine unterliegende Schicht nicht auf eine darüber liegende Schicht zugreifen darf. Werden die Schichten in geordneter Reihenfolge (oberste Schicht oben bzw. links) dargestellt, so lassen sich Verletzungen der Zugriffsregel direkt ablesen. Diese befinden sich in dem beschriebenen Fall im oberen, rechten Dreieck der Matrix.

3.7.2. Graphdarstellung

Da Traces üblicherweise gerichtet sind, lassen sich Traceabilityinformationen als gerichteter Graph [Die06] darstellen. Werkzeuge wie *YAKINDU Traceability* [Yak17a] ermöglichen es vor allem vertikale Traces (siehe Abschnitt 3.2) übersichtlich darzustellen. Dazu wird ein hierarchisches Layout (auch *Layered Drawings* [DKM06]) verwendet, sodass die verschiedenen Schichten erkennbar sind. Eine solche Darstellung aus YAKINDU Traceability wird in Abbildung 3.3 gezeigt.

Werden mit einer solchen Darstellung hingegen eine große Anzahl an Traces dargestellt, kann es zu vielen Überschneidungen kommen, was zur Unübersichtlichkeit führt. Dem kann

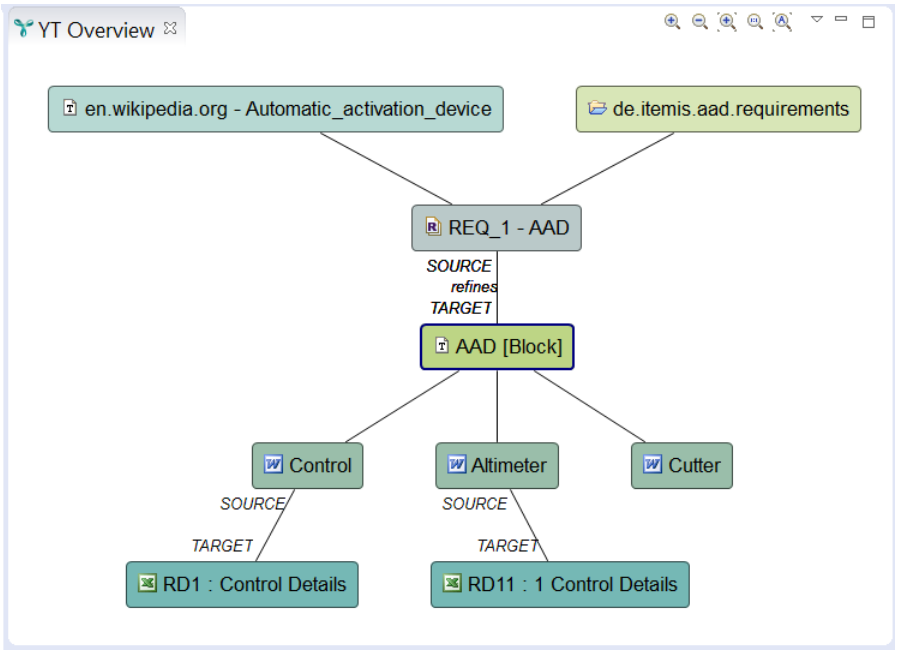

Abbildung 3.3.: Vertikale Tracebilityinformationen dargestellt als Graph unter Verwendung eines hierarchischen Layouts (entnommen von [Yak17b]).

durch die Auswahl einer Teilmenge und der Verwendung eines *Force-Directed-Layouts* (auch Force-Directed-Drawing [BM01]) abgeholfen werden. Bei dieser Darstellungsweise werden meist nur wenige Elemente ausgewählt, welche zentriert angezeigt werden. Zusätzlich wird die Nachbarschaft der ausgewählten Elemente angezeigt.

Im Falle des in Abbildung 3.4 dargestellten Werkzeugs, dem *Massey Architecture Explorer* [Die12], kann die Auswahl der aktiven Elemente interaktiv durch einen Mausklick im Graphen verändert werden. Somit eignet sich ein solches Werkzeug für eine manuelle Durchführung der in Abschnitt 3.4 beschriebenen Analysen, da komfortabel in Richtung der eingehenden und ausgehenden Links navigiert werden kann.

3.7.3. Softwarestadt

Die Visualisierung von Software als Softwarestadt bietet die Möglichkeit durch Metriken berechnete Zahlenwerte in einer interaktiven und navigierbaren 3D-Stadt darzustellen. Dabei sind in *CodeCity* [WL07] Grundfläche, Höhe, Farbe und Transparenz von Gebäuden jeweils Eigenschaften, welche mit einer Metrik oder mit anderen Nutzereinstellungen verknüpft werden können. Darüber hinaus können Paketstrukturen durch Ebenen dargestellt werden. Abbildung 3.5 zeigt eine beispielhafte Softwarestadt aus CodeCity. Durch diese Art der Visualisierung kann vor allem das Verständnis der dargestellten Software verbessert werden [WLR11].

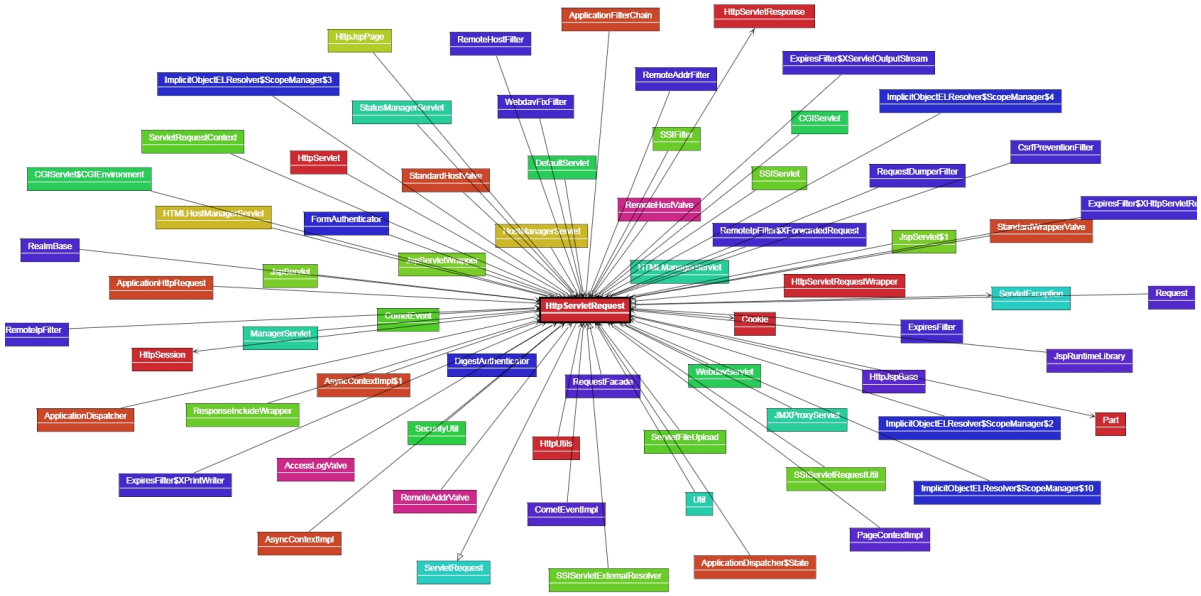

Abbildung 3.4.: Ausgewählte Elemente und ihre Nachbarschaft dargestellt als Graph unter Verwendung des Force-Directed-Layouts (erstellt mit [Mas17]).

Mit dem Werkzeug *Explore Viz* [FKH17] lassen sich Applikationslandschaften zur Laufzeit analysieren. Dabei lassen sich einzelne Applikationen ebenfalls als Softwarestadt darstellen. In dieser Stadtansicht werden Pakete im Gegensatz zu CodeCity ebenfalls als Gebäude dargestellt. Durch Doppelklick auf eine Gebäude wird die innere Sicht freigegeben, sodass Unterpakete oder einzelne Klassen – ebenfalls als Gebäude dargestellt – sichtbar werden. Darüber hinaus lässt sich die Kommunikation zwischen Klassen als Querbeziehungen in der Softwarestadt darstellen.

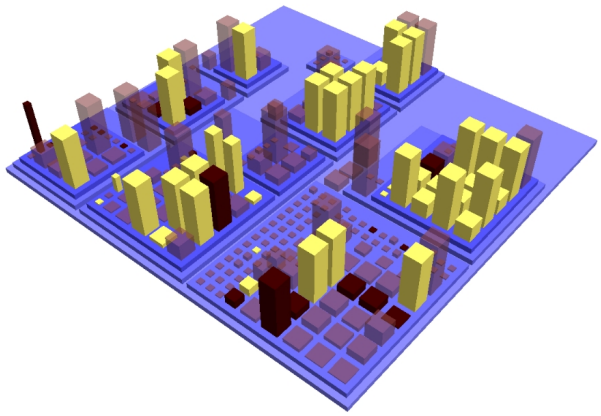

Abbildung 3.5.: Eine Softwarestadt aus dem Werkzeug *CodeCity* (entnommen aus [WL07]).

Die Softwarestadtdarstellung kann prinzipiell auch zu Darstellung von Softwarearchitekturen genutzt werden. Dazu müsste die Darstellung von Paketen durch die Darstellung von Architektureinheiten ausgetauscht werden.

3.7.4. Architekturvisualisierung

Architekturen lassen sich als verschachtelter Graph darstellen.

Definition 3.7.1 (Verschachtelter Graph)**.** *„G = (V, E) mit einer endlichen Knotenmenge V („Entitäten“), einer Kantenmenge E von Knotenpaaren („Zusammenhänge“) und einem Baum, dem Hierarchiebaum, dessen Blätter die Knoten in V sind. Jeder innere Knoten C des Baums repräsentiert die Knoten, die Blätter des Teilbaums mit Wurzel C sind. So definiert T eine Hierarchie von verschachtelten Teilmengen der Knoten [...].“[GJ14]*

Architektureinheiten bilden dabei die Menge C des Hierarchiebaums, die Artefakte bilden die Menge der Knoten V und deren Beziehungen die Menge der Kanten E des zugehörigen Graphen. Dargestellt werden die Elemente des Hierarchiebaums. Dabei lässt sich der Hierarchiebaum beliebig auf- oder zuklappen. Verbindungen werden jeweils zwischen den Elementen aus C dargestellt, die keine dargestellten Kinderknoten haben. Eine Verbindung wird zwischen solchen Elementen $c_1, c_2 \in C$, mit $c_1 \neq c_2$ dargestellt gdw. eine Kante $(a_1, a_2) \in E$ existiert, mit $a_1 \in c_1$ und $a_2 \in c_2$.

Beispiele für Werkzeuge, die die Architektur in Form eines verschachtelten Graphen darstellen sind HUSACCT [PKvdWB14] und ARAMIS [NLG+15]. Abbildung 3.6 zeigt einen zugehörigen Ausschnitt aus ARAMIS.

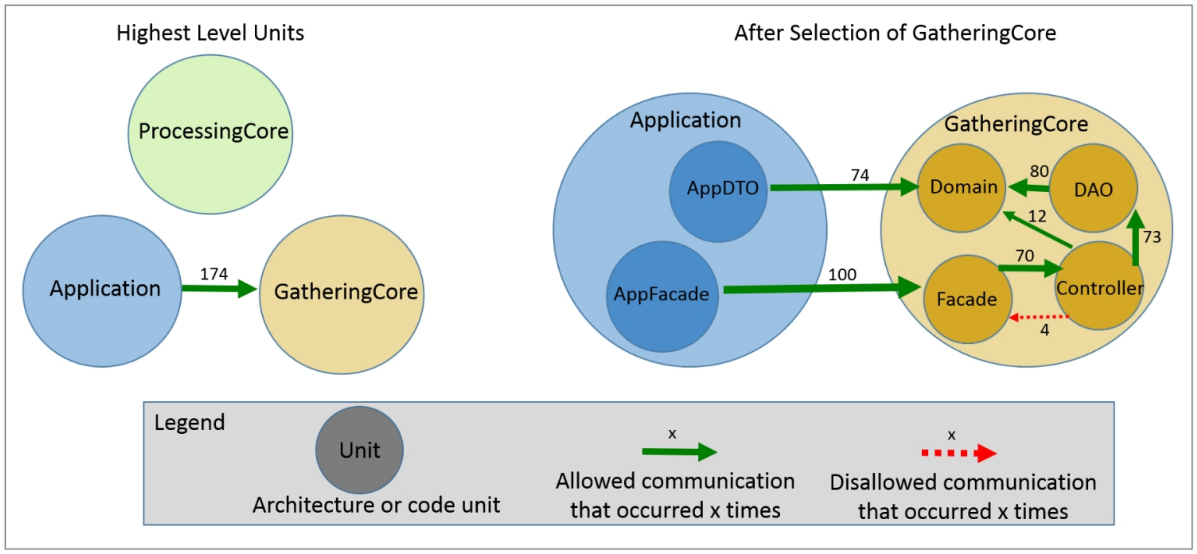

Abbildung 3.6.: Die *Circle-based hierarchical view* von ARAMIS zur Darstellung von Architekturbeziehungen (entnommen aus [NLG+15]).

Wird ein Element des Hierarchiebaums expandiert, so werden die beinhaltenden Elemente angezeigt. Beziehungen werden dabei nur zu den sichtbaren Knoten mit der größten Distanz zur Wurzel des Hierarchiebaums dargestellt, d.h. wird ein Element aufgeklappt, so wird die Beziehung dieses Elements aus der Visualisierung entfernt und stattdessen die Beziehungen der Kinderknoten dargestellt. Beim Schließen tritt dementsprechend der umgekehrte Fall ein. Neben den tatsächlich vorhandenen Beziehungen bei Ist-Architekturen oder der vorgesehenen Beziehungen bei Soll-Architekturen können Architekturverletzungen durch besonders eingefärbte Kanten oder Kanten mit besonderer Linienart dargestellt werden. Wie stark Architekturkomponenten gekoppelt sind wird oftmals durch die Linienstärke dargestellt, welche durch einen numerischen Wert auf der Kante ergänzt werden kann.

Darüber hinaus existiert eine weitere Möglichkeit einen verschachtelten Graphen darzustellen, welche im Werkzeug Sonargraph [Hel17a] Verwendung findet. Hier wird die Baumstruktur so sortiert, dass möglichst wenige Beziehungen von unten nach oben bestehen. Abbildung 3.7 zeigt die *Exploration View* des Werkzeugs.

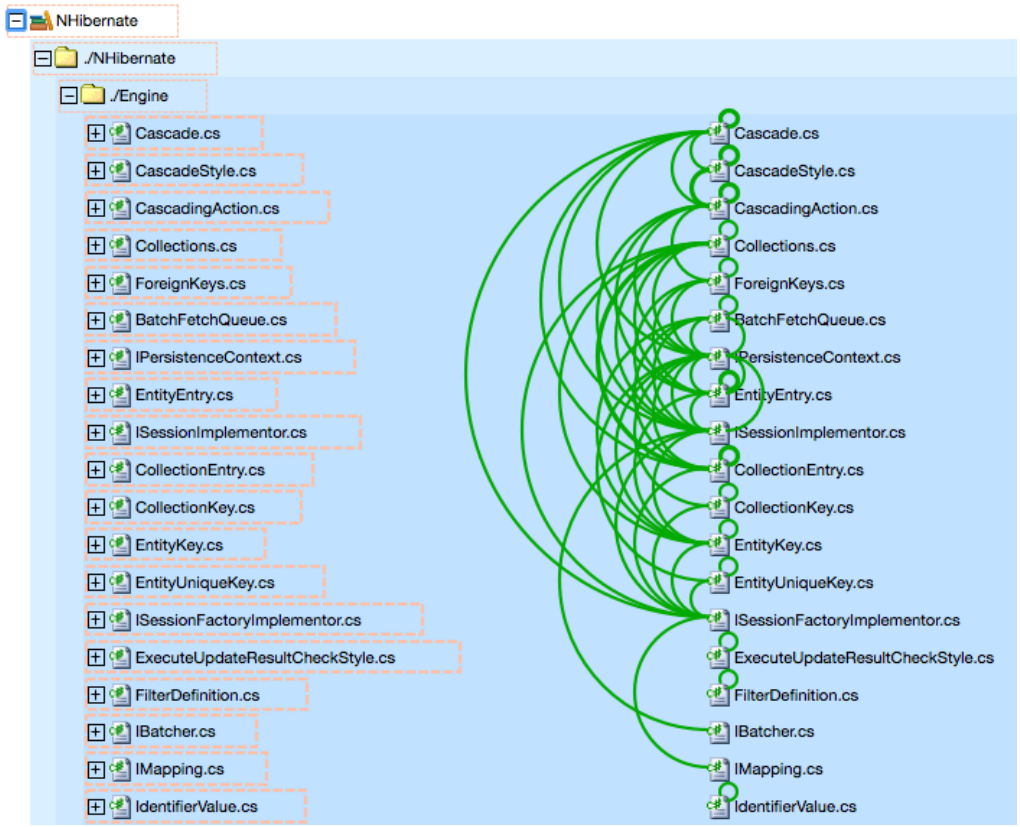

Abbildung 3.7.: Die *Exploration View* von Sonargraph zur Darstellung von Architekturbeziehungen (entnommen von [Hel17b]).

Beziehungen von oben nach unten werden über die Verbindungen links, Beziehungen von unten nach oben auf der rechten Seite der dargestellten Element angezeigt.

Kapitel 4.

Artefaktbasierte Analyse

Auf Basis der in Kapitel 2 vorgestellten Grundlagen der modellgetriebenen Softwareentwicklung und der in Kapitel 3 vorgestellten Analysetechniken für Softwaresysteme wird in diesem Kapitel eine Übersicht über das in dieser Arbeit entwickelte Gesamtkonzept zur Analyse von MDD-Projekten vorgestellt. Der in dieser Arbeit verwendete Ansatz setzt eine auf der UML/P basierende Modellierungstechnik (siehe Kapitel 5) ein, um die Artefakte und deren Beziehungen extrahieren und zugehörige Analysen spezifizieren zu können.

Bevor eine Übersicht über den entwickelten Lösungsansatz vorgestellt wird, werden zunächst der dafür eingeführte Artefaktbegriff erläutert (Abschnitt 4.1) und anhand eines Beispielszenarios Analysen auf Artefaktstrukturen weiter motiviert (Abschnitt 4.2). Anschließend folgt eine Übersicht über den Lösungsansatz (Abschnitt 4.3) und die daran beteiligten Rollen. Die Definition grundlegender, im Bezug zum Lösungsansatz stehender Begriffe folgt in Abschnitt 4.4. Darüber hinaus werden Anforderungen an die Modellierungstechnik, an die zu unterstützenden Analyseverfahren sowie an eine zugehörige Werkzeugunterstützung festgehalten (Abschnitt 4.5). Das in diesem Kapitel vorgestellte Szenario sowie der Lösungsansatz basieren auf den Arbeiten [BGRW17, BGRW18].

4.1. Artefakte in modellgetriebenen Softwareentwicklungsprojekten

Der Begriff *Artefakt* ist in der Literatur nicht eindeutig definiert, muss aber für diese Arbeit präzise definiert werden, um den Fokus der Analysemethodik einordnen zu können. Im Folgenden wird eine Auswahl von Definitionen aus der Literatur vorgestellt und diskutiert. Anschließend wird eine Definition des Artefaktbegriffs gegeben, welcher in dieser Form durchgängig in der vorliegenden Arbeit verwendet wird.

- *„In a model-based design and development paradigm, however, an artefact is seen as a structured abstraction of modelling elements used as input, output, or as an intermediate result of a process.“* [FPKB10]

- *„Ein **Artefakt** (artifact) ist in der UML etwas physisch Vorhandenes, z. B. eine Datei auf der Festplatte.“* [Bal11]
- *„Entwicklungsergebnisse werden in einer konkreten Notation dargestellt. Dafür werden zum Beispiel die natürliche Sprache, UML oder eine Programmiersprache verwendet. Die Dokumente dieser Sprachen werden Artefakte genannt. Dies sind beispielsweise Anforderungsanalysen, Modelle, Code, Reviewergebnisse oder ein Glossar. Ein Artefakt kann hierarchisch gegliedert sein.“* [Rum12]
- *„An artifact [. . .] is defined as any (tentative) deliverable that is created, consumed, or modified by an activity.“* [KB14]
- *„An artifact is the documentation of a result of a step of system development in software or systems engineering. There are many instances of artifacts such as data models, requirements documents, software or system specifications, architecture descriptions, program modules, test cases, and quality requirements.“* [Bro18]

Während zwei der Definitionen vor allem auf die physische Manifestation abzielen, ist in den anderen Definitionen die Rolle im Entwicklungsprozess entscheidend. In [Bro18] werden Artefakte zusätzlich durch die folgenden Eigenschaften unterschieden:

- ihren logischen Inhalt
- die Repräsentation des logischen Inhalts
- ihre physische Repräsentation

Somit ist auch nach [Bro18] die physische Repräsentation zwar nicht das einzige, jedoch ein entscheidendes Merkmal von Artefakten. In dieser Arbeit wird die folgende Definition für Artefakte verwendet.

Definition 4.1.1 (Artefakt)**.** *Ein Artefakt ist eine individuell speicherbare Einheit mit einem eindeutigen Namen, die einem bestimmten Zweck im Kontext des Software-Engineering-Prozesses dient.*

In dieser Definition wird der Fokus eher auf die physische Manifestation des Artefakts, als auf die Rolle im Entwicklungsprozess gelegt. Dabei wird gefordert, dass ein Artefakt sowohl als individuelle Einheit gespeichert als auch anschließend referenziert werden kann. Trotzdem sollte ein Artefakt im Zuge des Entwicklungsprozesses einem definierten Zweck dienen, da die Erstellung und Wartung des Artefakts ansonsten unnötig wäre. Allerdings wird nichts darüber ausgesagt, wie das Artefakt in den Entwicklungsprozess eingebunden ist, d.h., ein Artefakt muss nicht zwangsläufig das Zielsystem, zum Beispiel in Form eines Modells, beschreiben, sondern kann stattdessen auch Teil der MDD-Werkzeugkette sein. Der logische Inhalt von Artefakten bleibt somit weitestgehend unbeachtet. Dieser Grad an Abstraktion wurde gewählt,

um effektiv Analysen der Artefaktstruktur unter Betrachtung der bestehenden heterogenen Beziehungen durchführen zu können und damit dem Fokus dieser Arbeit gerecht zu werden. Wie später in Kapitel 6 gezeigt wird, ist die Repräsentation des logischen Inhalts zumindest ein Stück weit durch die jeweilige Artefaktart vorgegeben.

Typische Artefakte sind Dateien, wie zum Beispiel Java- oder Modelldateien. Definition 4.1.1 schließt dabei aber auch anderweitige Artefakte mit ein. So ist eine Persistierung in Datenbanken ebenfalls denkbar. In der Java-Spezifikation [GJS+15] wird beispielsweise die Möglichkeit der Speicherung von Java-Paketen und *Compilation Units* sowohl in Form von Dateien, als auch in Datenbanken erwähnt. In dieser Arbeit wird vor allem auf die Speicherung in Dateiform fokussiert.

Auch die Beziehungen zwischen Artefakten müssen definiert werden. Die Granularität, in der zwischen Artefaktbeziehungen unterschieden wird, variiert zum Teil stark. In [HSG11] wird beispielsweise zwischen drei Beziehungsarten zwischen heterogenen Modellen unterschieden während in [PvdW15] alleine für Java-Typen zwischen sieben verschiedenen Abhängigkeitsarten und unter Berücksichtigung der Unterabhängigkeitsarten sogar insgesamt zwischen 31 Beziehungsarten unterschieden wird.

Aus diesem Grund sollte sich der Abstraktionsgrad, in dem zwischen Beziehungsarten unterschieden wird, je nach Anwendungsfall bestimmen lassen. Dies wird in Abschnitt 4.5 als Anforderung *MRQ-3.2* festgehalten und von dem in dieser Arbeit entwickelten Ansatz unterstützt, indem sich der Abstraktionsgrad von Beziehungsarten projektspezifisch definieren lässt. Dazu wird auf der abstraktesten Ebene zwischen den drei Arten von Artefaktbeziehungen *beinhaltet*, *bezieht sich auf* und *produziert* unterschieden (vgl. Abschnitt 3.2), die sich dann projektspezifisch spezialisieren lassen.

Basierend auf dieser Sichtweise auf Artefakte und deren Beziehungen wird in den folgenden Abschnitten zunächst ein motivierendes Beispiel und anschließend ein Überblick über den gewählten Lösungsansatz vorgestellt.

4.2. Szenario

Wie in Abschnitt 2.1 beschrieben hilft die modellgetriebene Softwareentwicklung dabei, die Komplexität bei der Softwareentwicklung zu beherrschen. Gerade bei sehr großen und komplexen MDD-Projekten besteht die Herausforderung, dass die neu eingeführten Werkzeuge und Artefakte, wie beispielsweise Konfigurationsdateien, Modelle, Templates, Transformationen und ihre Beziehungen, ebenfalls verstanden und gehandhabt werden müssen. Insbesondere dann wenn Modelle aus unterschiedlichen Abteilungen und Domänen [CCF+15a], an unterschiedlicher Stelle im Entwicklungsprozess, zur Entwicklung verschiedener Produkte einer Softwareproduktline [PBvdL05, RSW+15, RRS+16] eingesetzt werden, handgeschriebener Quellcode mit generiertem Code zusammenarbeiten muss [GHK+15a, GHK+15b, BRW17] oder Generatoren nach ihrer Auslieferung stark konfiguriert werden können [GMR+16, Rot17], wird diese Situation komplex und unübersichtlich. Ein einfaches Verständnis der Projektsituati-

on mit Fokus auf die verwendeten Werkzeuge und beteiligten Artefakte, die auf vielfältige Art und Weise in Beziehung stehen, ist so nicht mehr möglich. Das in Abbildung 4.1 dargestellte Beispielszenario soll diese Problematik verdeutlichen.

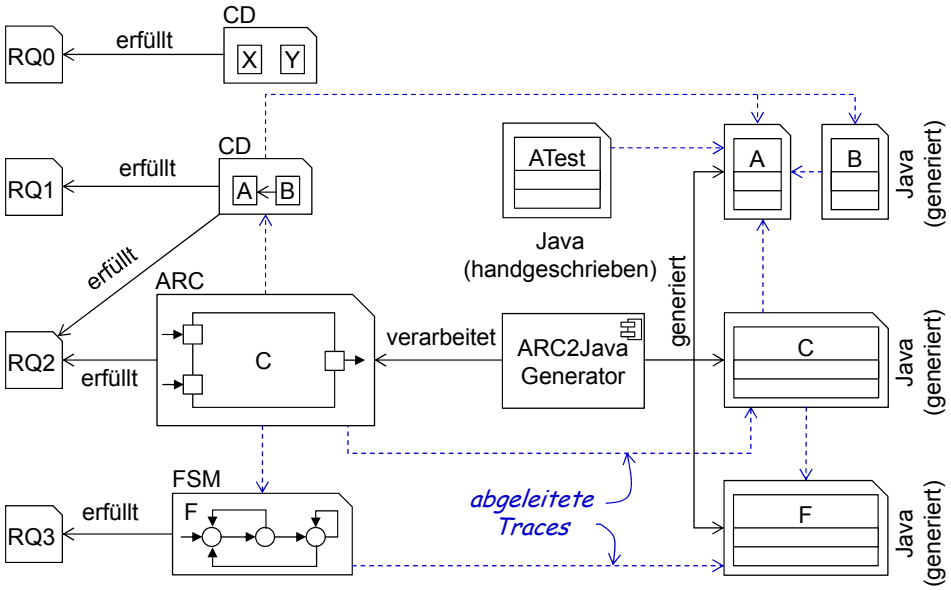

Abbildung 4.1.: Beispielszenario eines Softwareentwicklungsprojekts mit MAA (angepasst aus [BGRW18]).

Das Beispielszenario zeigt die in einem MDD-Projekt vorhandenen Artefakte sowie ein einzelnes Werkzeug (`ARC2JavaGenerator`). Der zugrunde liegende Prozess könnte hier wie folgt aussehen:

Als erstes spezifiziert ein Anforderungsentwickler Anforderungen (RQ0, . . . , RQ3) an das zu entwickelnde Softwareprodukt. In der darauffolgenden Phase werden Modelle, die die Anforderungen reflektieren, von Modellierern erstellt. Jede Anforderung definiert Bedingungen, die durch ein oder mehrere Modellartefakte erfüllt werden. Modellartefakte können dabei zur Erfüllung mehrerer Anforderungen beitragen und werden strikt nach der zugehörigen Modellierungssprache erstellt. Letzteres ist wichtig, damit sie in dem nächsten Prozessschritt automatisiert weiterverarbeitet werden können. In diesem Szenario wird ein Architektur-zu-Java-Generator eingesetzt. Wie in Abschnitt 2.4.2 erläutert, unterstützt die Sprachfamilie *MontiArcAutomaton* Klassendiagramme (CD), Architekturmodelle (ARC) sowie endliche Automaten (FSM). Die Traces zur Darstellung der Erfüllung einer Anforderung durch Modellartefakte werden von den Modellierern manuell hinterlegt.

Im nächsten Prozessschritt wird das MDD-Werkzeug ausgeführt, welches aus den gegebenen, heterogenen Modellen Java Code generiert. In diesem Beispiel wird der generierte Code durch einen handgeschriebenen Test (`ATest`) abgesichert.

Die in Abbildung 4.1 dargestellten durchgezogenen Pfeile stellen die Beziehungen zwischen Werkzeugen und Artefakten dar, die direkt durch den Prozess offengelegt werden. Dem liegt

die Annahme zugrunde, dass sich das primäre Eingabe-ARC-Modell eindeutig identifizieren lässt und die durch den Generator erstellten Java-Klassen durch ihren Ablageort als generierte Klassen des zu erstellenden Produkts erkannt werden können.

Die durch gestrichelte Pfeile dargestellten Beziehungen ergeben sich hingegen nicht direkt. Wie in Abschnitt 3.2 vorgestellt, ist das Verständnis der Zuordnung von Eingabe- und Ausgabeartefakten eines oder mehrerer Prozessschritte ein übliches Problem in großen MDD-Projekten. Auch bestehen zwischen den verschiedenen Modellartefakten einer Modellierungssprachfamile verschiedene Arten von Beziehungen. Um solche Beziehungen nachvollziehen und verstehen zu können, wird sprachspezifisches Wissen benötigt. Ähnliches gilt für die Beziehungen zwischen Java-Artefakten. Während hier bereits durch existierende Werkzeuge (siehe Abschnitt 3.3.1) einfacher Abhilfe geschaffen werden kann, ist dies für andere Artefakte und die beteiligten Werkzeuge oft nicht der Fall. Weiterhin ist normalerweise eine integrierte Betrachtung aller heterogenen Beziehungen nicht möglich.

Ein Teilziel dieser Arbeit ist es, solche Beziehungen zu explizieren, damit eine durchgängige Betrachtung und Analyse erfolgen kann. Für das Beispiel ergeben sich dadurch bereits die folgenden verschiedenen Möglichkeiten, ein Tracing zwischen den Artefakten herzustellen oder Change-Impact-Analysen durchzuführen. Nach der Extraktion der zusätzlichen, in Abbildung 4.1 gestrichelt dargestellten Beziehungen können diese beispielsweise dazu genutzt werden, um

1. auszuwerten, welche generierten Artefakte notwendig sind, um bestimmte Anforderungen zu erfüllen. Diese Informationen können durch Betrachtung der Beziehungen zwischen Modellen und Anforderungen sowie der Beziehungen zwischen Modellen und daraus generiertem Code abgeleitet werden.

2. auszuwerten, welcher Testfall welche Anforderung verifiziert. Dazu kann die Methode aus (1.) genutzt werden, wobei zusätzlich die Beziehungen zwischen Tests und generiertem Code mit in Betracht gezogen werden müssen.

3. auszuwerten, ob ungenutzte Modelldateien, welche zu unnötiger Komplexität in dem MDD-Projekt führen, existieren. Solche Modelle können durch die Zuordnung zwischen Eingabe- und Ausgabeartefakten identifiziert werden. Gibt es keine Zuordnung von mindestens einem Codeartefakt für ein Modellartefakt ist dieses Modellartefakt ein Kandidat für eine Entfernung. In dem gezeigten Beispiel ist das oberste CD-Artefakt ein solcher Kandidat.

4. auszuwerten, ob ungenutzte Codedateien existieren. Dies ist in dem gezeigten Beispiel für die generierte Klasse `B` der Fall, da diese zwar Klasse `A` nutzt, jedoch selbst von keiner anderen Klasse genutzt wird. Die Klasse `C` wird hier als Ausgangspunkt für die Untersuchung festgelegt, da sie direkt durch das primäre ARC-Modell generiert wurde, was sie in diesem Fall als Hauptklasse des generierten Programms kennzeichnet. Durch das Aufspüren von ungenutzten Codedateien, welche nicht weiter getestet, gepackt

oder ausgeliefert werden müssen, kann die Effizienz des MDD-Entwicklungsprozesses gesteigert werden.

5. auszuwerten, ob die durch den Generator durchgeführte M2T-Transformation alle Anforderungen erfüllt. Hierzu kann wiederum die Methode aus (3.) verwendet werden. Existieren Anforderungen, welche ausschließlich durch ungenutzte Modelle repräsentiert sind, werden diese nicht von der Transformation erfüllt. In dem gezeigten Beispiel ist dies für `RQ0` der Fall.

6. den Aufwand für Änderungen abzuschätzen. Durch die vorhandenen Traces kann berechnet werden, welche Artefakte potentiell von der Änderung eines Artefakts betroffen sind (siehe Abschnitt 3.4.2). Die Güte einer solchen Abschätzung hängt von dem Detailgrad ab, indem Beziehungsarten unterschieden werden.

4.3. Lösungsansatz

In diesem Abschnitt wird eine Übersicht über den in dieser Arbeit entwickelten Lösungsansatz und die beteiligten Rollen gegeben. Mit Hilfe des Lösungsansatzes lassen sich artefaktbasierte Analysen, wie die in Abschnitt 4.2 vorgestellten beispielhaften Analysen, durchführen. Die Rollen werden im weiteren Verlauf der Arbeit zur Erklärung der einzelnen Teile des Gesamtansatzes verwendet. Aus Gründen der Vereinfachung wird ausschließlich die männliche Form des Rollennamens verwendet.

Ein wichtiger Bestandteil des Ansatzes ist es, die verschiedenen Betrachtungsgegenstände, also Artefakte, Werkzeuge, Systeme, etc. und ihre Beziehungen explizit zu machen. Dabei werden Modellierungstechniken verwendet, welche es ermöglichen, die Betrachtungsgegenstände und deren Beziehungen zu prüfen und somit weiter für modellbasierte Analysen nutzen zu können. Abbildung 4.2 gibt einen Überblick über das entwickelte modellbasierte Lösungskonzept.

Der Ansatz sieht vor, dass zunächst explizit festgelegt wird, welche Arten von Artefakten, Werkzeugen, sonstigen Elementen und Beziehungen in dem zu untersuchenden MDD-Projekt von Interesse sind. Dies wird von dem *Architekt* des gesamten MDD-Projekts durch die Modellierung eines *Artefaktmodells* (AM) vorgenommen. Ein solches Modell strukturiert somit das entsprechende MDD-Projekt. Da dort die Arten der Betrachtungsgegenstände und Beziehungen und nicht die konkreten Instanzen modelliert werden, kann ein solches Modell theoretisch über den gesamten Projektlebenszyklus unverändert genutzt werden. In der Praxis können jedoch auch neue Artefakt- oder Beziehungsarten im Laufe der Zeit hinzukommen oder wegfallen. Dies kann beispielsweise durch einen Wechsel der verwendeten MDD-Werkzeuge hervorgerufen werden.

Zu dem AM passende Artefaktdaten, die den aktuellen Projektzustand widerspiegeln, lassen sich idealerweise automatisch extrahieren und in einem oder mehreren Artefakten abspeichern. Diese Daten stehen in einer ontologischen Instanzbeziehung [AK03] zu dem AM, wodurch sich jedes Element und jede Beziehung aus den Artefaktdaten einer Element- bzw. Beziehungsart

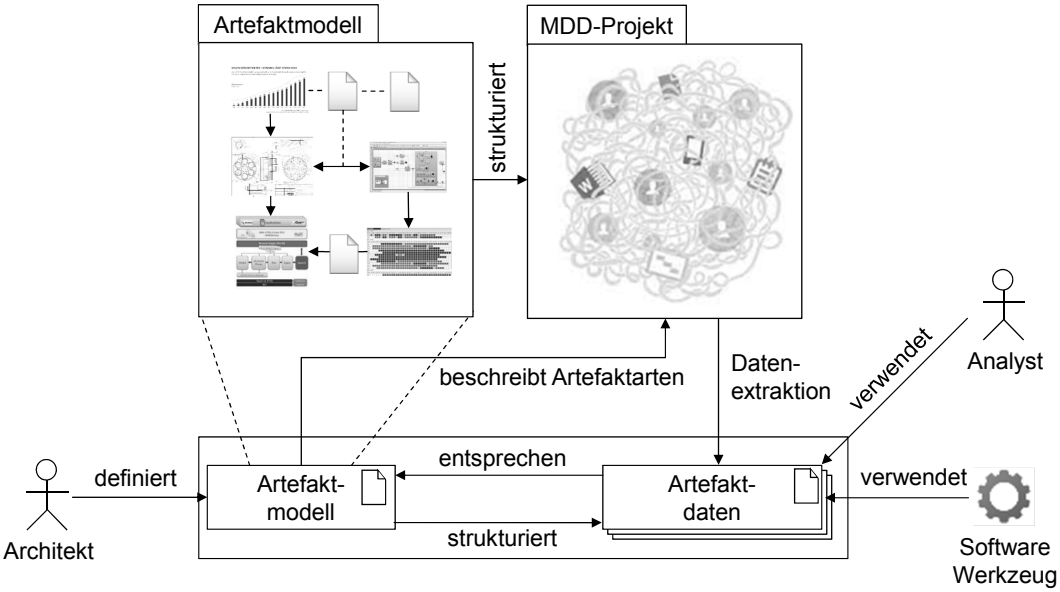

Abbildung 4.2.: Lösungsansatz zur artefaktbasierten Analyse modellgetriebener Softwareentwicklungsprojekte (angepasst aus [BGRW18]).

des AMs zuordnen lässt. Somit gibt das AM die Struktur der Artefaktdaten vor. Ein zusammenhängender Artefaktdatensatz repräsentiert den Projektzustand zu einem bestimmten Zeitpunkt. *Analysten* oder spezielle Analysewerkzeuge können dann die Artefaktdaten verwenden, um einen Überblick über den Projektzustand zu bekommen, bestimmte Sachverhalte zu prüfen, Berichte zu erstellen und vorhandene Optimierungspotenziale im Projekt offen zu legen. Das übergeordnete Ziel ist es, den modellgetriebenen Entwicklungsprozess so effizient wie möglich zu gestalten. Ein ähnlicher Ansatz zur Erhebung und Bereitstellung von Daten wird in [CNSM13] beschrieben. Der Ansatz sieht allerdings keine explizite Modellierung eines AMs vor.

Wie später in Abschnitt 5.2 und Kapitel 6 gezeigt wird, lassen sich durch ein solches AM und die zugehörigen Artefaktdaten unter anderem die Organisation der Artefakte im Dateisystem, die Abläufe einer Werkzeugkette und statische Kenntnisbeziehungen zwischen heterogenen Artefakten abbilden. Dadurch entsteht eine Übersicht der Projektarchitektur unter Betrachtung von Eingabemodellen, MDD-Werkzeugen, welche selbst aus Artefakten bestehen sowie handgeschrieben oder generierten Artefakten, die zum Zielprodukt des Entwicklungsprozesses gehören. Ein solches Modell hängt somit von den im Projekt verwendeten Sprachen, Werkzeugen und Technologien ab und muss oftmals projektspezifisch zugeschnitten werden. Für ähnlich Projekte lässt sich ein solches Modell aber gänzlich oder zumindest in Teilen wiederverwenden.

Neben den in Abbildung 4.2 bereits dargestellten Rollen existieren weitere Rollen und Spezialisierungen von Rollen. Alle in dieser Arbeit verwendeten Rollen werden im Folgenden vorgestellt. Dabei wird unterschieden, ob es sich bei der Rolle um eine im MDD-Projekt beteiligte Rolle, eine an der Analyse beteiligte Rolle oder eine übergreifende Rolle handelt.

Rollen des MDD-Projekts

- Der *Werkzeugentwickler* entwickelt MDD-Werkzeuge, die in dem Projekt zur Erstellung des Zielprodukts genutzt werden. Der Werkzeugentwickler umfasst in dieser Arbeit mehrere der in [KRV06] genannten Rollen: *Sprachentwickler*, *Werkzeugentwickler* und *Bibliotheksentwickler*. Auch der in [Höl18] genannte *Transformationswerkzeugentwickler* wird in dieser Arbeit in die Rolle des Werkzeugentwicklers mit eingeschlossen.

- Der *Entwickler* erzeugt mit Hilfe der vom *Werkzeugentwickler* bereitgestellten MDD-Werkzeuge das Zielprodukt. Dabei kann als Erweiterung zu [KRV06] zwischen zwei spezialisierten Rollen unterschieden werden:
  - Der *Modellierer* erstellt die im Projekt verwendeten Modelle, welche das zu entwickelnde Produkt beschreiben und gleichzeitig die Eingabe für die verwendete MDD-Werkzeugkette sind.
  - Der *Programmierer* passt das teilweise generierte Produkt durch die Ergänzung von handgeschrieben Code an.

Analyserollen

- Der *Analyst* entwickelt projektspezifische Artefaktanalysen. Seine Tätigkeit umfasst die konzeptionelle Entwicklung der Analysen, deren modellbasierte Spezifikation sowie deren Durchführung unter Verwendung spezieller Analysewerkzeuge.

- Der *Analysewerkzeugentwickler* entwickelt projektspezifische Analysewerkzeuge oder passt existierende Werkzeuge an projektspezifische Bedarfe an. Die entwickelten Analysewerkzeuge können wie oben beschrieben vom *Analyst* zur Durchführung spezifischer Analysen verwendet werden.

Übergreifende Rollen

- Der *Architekt* strukturiert das MDD-Projekt, d.h., er ist für die Organisation der Artefakte und Werkzeuge und deren Zusammenspiel im Projekt verantwortlich. Darüber hinaus spezifiziert der Architekt ein projektspezifisches AM, welches in dem in dieser Arbeit entwickelten Lösungskonzept Ausgangspunkt für die Spezifikation und Durchführung von Analysen ist.

- Der *Administrator* verwaltet die für die Durchführung des MDD-Projekts benötigte Software-Engineering-Infrastruktur (siehe Abschnitt 9.5.1). Darüber hinaus wird eine entsprechende Infrastruktur ebenfalls benötigt, um Analysen in regelmäßigen Abständen oder bei Eintreten bestimmter Ereignisse, wie beispielsweise Projektänderungen, automatisiert durchführen zu können (siehe Abschnitt 9.5.2). Weiterhin administriert er projektübergreifend verwendete Analysewerkzeuge (siehe Abschnitt 8.5.5).

4.4. Grundlegende Begriffe

Nachdem in Abschnitt 4.1 bereits der Artefaktbegriff definiert und in Abschnitt 4.3 ein Überblick über den Lösungsansatz gegeben sowie die beteiligten Rollen vorgestellt wurden, werden in diesem Abschnitt weitere, grundlegende und mit dem Lösungsansatz in Verbindung stehende Begriffe definiert.

Wie in Abbildung 4.2 dargestellt, wird ein AM dazu genutzt, die beteiligten Elemente des zu analysierenden MDD-Projekts zu strukturieren. In dieser Arbeit wird die folgende Definition verwendet:

Definition 4.4.1 (Artefaktmodell)**.** *Das Artefaktmodell definiert die relevanten Artefaktarten sowie zugehörige Beziehungsarten des zu untersuchenden MDD-Projekts.*

Der Fokus liegt somit auf der Definition der Artefaktarten. Es können jedoch auch andere Elementarten aus MDD-Projekten, wie beispielsweise Pakete, Typen oder Module, modelliert werden. Beziehungen werden in dieser Arbeit nicht als Elemente bezeichnet. In der UML wird bereits ein Vorschlag für relevante Artefaktarten gemacht. Durch Nutzung von Stereotypen lässt sich zwischen den folgenden Artefaktarten unterschieden: *file*, *document*, *executable*, *library*, *script* und *source* [Bal11]. Eine solche Einteilung in Artefaktarten kann als Ausgangspunkt für ein AM dienen. Zusätzlich werden die möglichen Beziehungsarten zwischen den Artefakten definiert. In dieser Arbeit wird nicht von einem universell verwendbaren AM für alle Projekte ausgegangen. Stattdessen wird als Teil der Methodik die Erstellung projektspezifischer AMe (siehe Abschnitt 7.1) vorgesehen. Dies hat den Vorteil, dass spezifischere Beziehungen zwischen Artefakten modelliert und Analysen deshalb genauer auf Projekte zugeschnitten werden können. Bei dieser Methodik kann auf einen wiederverwendbaren AM-Kern (siehe Abschnitt 5.2) zurückgegriffen werden.

Der zweite Begriff, der in Abbildung 4.2 bereits verwendet wurde, ist der Begriff *Artefaktdaten*. In dieser Arbeit wird für diesen Begriff die folgende Definition verwendet:

Definition 4.4.2 (Artefaktdaten)**.** *Artefaktdaten enthalten Informationen über die relevanten Artefakte und deren Beziehungen, die zu einem bestimmten Zeitpunkt in dem zu untersuchenden MDD-Projekt vorhanden sind. Sie sind bezüglich eines Artefaktmodells definiert.*

Nach dieser Definition wird genau ein Zeitpunkt betrachtet. Dabei können Artefaktdaten auch Daten über den Ablauf der im MDD-Projekt verwendeten Werkzeugkette enthalten. In diesem Fall können alle relevanten Aktionen, die bei Ablauf der Werkzeugkette bis zu dem gewählten Zeitpunkt bereits durchgeführt wurden, in den Artefaktdaten enthalten sein. Die Struktur der Artefaktdaten wird durch ein zugehöriges Artefaktmodell vorgegeben.

Basierend auf diesen beiden Definitionen lässt sich nun auch der Begriff *artefaktbasierte Analyse* präziser definieren. Wie in Abschnitt 1.2 beschrieben, ist die Möglichkeit zur Durchführung von Analysen der Artefaktstruktur zu schaffen eines der zentralen Ziele dieser Arbeit.

Definition 4.4.3 (Artefaktbasierte Analyse)**.** *Die artefaktbasierte Analyse bezeichnet alle Schritte, die notwendig sind, um ausgehend von einem MDD-Projekt und einem zugehörigen Artefaktmodell spezifizierte Analysen mit dem Ziel des Verständnisses, der Korrektur oder der Optimierung des untersuchten Projekts, durchzuführen.*

Somit wird unter diesem Begriff die Durchführung mehrerer Schritte zusammengefasst. Dies beinhaltet insbesondere auch die Extraktion von Artefaktdaten. Wie in Abschnitt 3.3 beschrieben, wird in dieser Arbeit zwischen der Extraktion von Artefaktdaten und deren Analyse unterschieden. Für letztere wird in dieser Arbeit der Begriff *Artefaktdatenanalyse* verwendet.

Definition 4.4.4 (Artefaktdatenanalyse)**.** *Die Artefaktdatenanalyse bezeichnet den Teilschritt der artefaktbasierten Analyse, bei dem die zuvor erhobenen Artefaktdaten analysiert werden.*

Diese Definition wird insbesondere bei Vorstellung der entwickelten Methodik in Kapitel 7 genutzt. Dort lassen sich auch die weiteren Schritte der artefaktbasierten Analyse finden.

Bei der Spezifikation des Artefaktmodells, der Extraktion von Artefaktdaten und der Spezifikation von Artefaktdatenanalysen werden Modelle erstellt. Der Begriff der *Artefaktmodellierung* fasst die zugehörigen Modellierungstätigkeiten wie folgt zusammen.

Definition 4.4.5 (Artefaktmodellierung)**.** *Die Artefaktmodellierung umfasst alle Modellierungstätigkeiten, die bei artefaktbasierten Analysen anfallen oder für diese notwendig sind.*

Somit wird die Modellbildung von allen im Rahmen des Lösungsansatzes benötigten Modellen unter dem Begriff der Artefaktmodellierung zusammengefasst. Dabei wird nicht unterschieden, ob die Modellbildung manuell durch den *Architekt* oder *Analyst* oder automatisiert durch Verwendung von MDD-Werkzeugen stattfindet. In welcher Modellierungssprache die einzelnen Modelle erstellt werden und wie diese kombiniert werden können wird durch die in Abschnitt 5.1 definierte Modellierungstechnik festgelegt.

4.5. Anforderungen

Aus dem vorgestellten Lösungsansatz ergeben sich verschiedene Anforderungen an die Modellierungstechnik zur Artefaktmodellierung und an eine Werkzeugunterstützung für die artefaktbasierte Analyse. Weiterhin werden Anforderungen an die zu unterstützenden Analyseverfahren festgehalten, die sich aus den relevanten Verfahren aus Kapitel 3 ergeben. Bei der Definition der Anforderungen sind die Erfahrungen und Vorarbeiten des Lehrstuhls für Software Engineering mit eingeflossen.

Die in diesem Abschnitt beschriebenen Anforderungen werden in späteren Abschnitten referenziert, z.B. dann wenn eine der Anforderungen durch das in dem Abschnitt vorgestellt Konzept erfüllt werden kann. Zu diesem Zweck wurde jeder Anforderung ein eindeutiger Identifikator zugewiesen. Um für eine Anforderung alle relevanten Abschnitte zu finden, können die Tabellen in Anhang B genutzt werden, die ein Tracing der Anforderungen zu den entsprechenden Abschnitten erlauben.

4.5.1. Anforderungen an Analyseverfahren

Die in Kapitel 3 vorgestellten Analyseverfahren lassen sich oftmals auf die Analyse von MDD-Projekten übertragen. Vorab wurden besonders interessante Analyseverfahren und Ansätze bewertet und als Analyseanforderungen (ARQs) für den Ansatz zur artefaktbasierten Analyse festgehalten.

ARQ-1 **Extraktionsverfahren:** Es sollen sowohl dynamische als auch statische Extraktionsverfahren unterstützt und deren Ergebnisse kombiniert in Analysen betrachtet werden können.

ARQ-2 **Change-Impact-Analyse:** Es sollen Change-Impact-Analysen durchgeführt werden können.

ARQ-3 **Abhängigkeitsanalyse:** Abhängigkeitsanalysen (vgl. Abschnitt 3.4.1) sollen durchgeführt werden können.

ARQ-4 **Ist-Architektur:** Die Rekonstruktion der Ist-Architektur des MDD-Projekts soll ermöglicht werden. Architektureinheiten sollen dabei beliebige Artefakte zugewiesen werden können.

ARQ-5 **Differenzarchitektur:** Die Berechnung von Differenzarchitekturen (vgl. Abschnitt 3.5) für MDD-Projekte soll durchgeführt werden können.

ARQ-6 **Metriken:** Es soll die projektspezifische Spezifikation und Berechnung von Metriken unterstützt werden.

4.5.2. Anforderungen an die Modellierungstechnik

Der in Abschnitt 4.3 skizzierte Lösungsansatz basiert auf der Verwendung einer Modellierungstechnik zur Spezifikation eines AMs und projektspezifischer Analysen sowie einem passenden Datenformat für Artefaktdaten. Konkrete Anforderungen an die Modellierungstechnik (MRQs) zur Artefaktmodellierung sind dabei:

MRQ-1 **Wiederverwendung von Modellierungssprachen:** Vorhandene Modellierungssprachen und die zugehörige Werkzeugunterstützung sollen nach Möglichkeit wiederverwendet werden.

MRQ-2 **Schlanke Modellierung:** Die Lesbarkeit der Modelle soll nicht durch die Überfrachtung unnötiger Informationen beeinträchtigt werden. Stattdessen soll es möglich sein, Zusatzinformationen in separate Artefakte auszulagern.

MRQ-3 **Erweiterbarkeit des Artefaktmodells:** Es soll komfortabel möglich sein Teile des Artefaktmodells wiederzuverwenden und das Modell zu erweitern.

MRQ-3.1 Modellkern: Insbesondere soll ein wiederverwendbares Kernmodell existieren, welches die in dieser Arbeit eingenommene Sichtweise auf Artefakte (vgl. Abschnitt 4.1) repräsentiert.

MRQ-3.2 Verfeinerung Der Abstraktionsgrad in dem zwischen Beziehungs- und Artefaktarten unterschieden wird, soll sich je nach Projekt definieren lassen. Dabei soll es möglich sein mehrere Abstraktionsstufen in einem Modell zu verwenden.

MRQ-4 Umfang des Artefaktmodells: Struktur (*bezieht-sich-auf* - und *beinhaltet*-Beziehungen, vgl. Abschnitt 3.2) und Abläufe (*produziert*-Beziehungen, vgl. Abschnitt 3.2) müssen sich durch das Artefaktmodell modellieren lassen, um Analysen der Struktur und Abläufe in Kombination betrachten zu können.

MRQ-5 Lesbarkeit von Artefaktdaten: Artefaktdaten sollen in einer für Menschen möglichst gut verständlichen Form spezifiziert werden, um auch manuelle Analysen ohne weitere Werkzeugunterstützung zu ermöglichen.

MRQ-6 Maschinelle Verarbeitung von Artefaktdaten: Das Format in dem Artefaktdaten spezifiziert werden, muss für eine maschinelle Weiterverarbeitung durch MDD-Werkzeuge geeignet sein.

MRQ-7 Analysespezifikation: Die präzise Spezifikation von projektspezifischen Artefaktdatenanalysen basierend auf dem AM soll möglich sein.

MRQ-8 Komfortable Modellierung: Die manuelle Erstellung von Modellen soll möglichst einfach und komfortabel sein.

MRQ-9 Konformität von Artefaktmodell und Artefaktdaten: Eine Konformitätsprüfung von Artefaktdaten zu dem Artefaktmodell soll möglich sein, d.h., es muss für jedes Element und jede Beziehung aus den Artefaktdaten möglich sein zu prüfen, ob diese der Spezifikation des zugehörigen Artefaktmodells entsprechen.

4.5.3. Anforderungen an die Werkzeugunterstützung

Neben den Anforderungen an zu unterstützende Analyseverfahren und die Modellierungstechnik werden Anforderungen an eine Werkzeugunterstützung (TRQs) für die artefaktbasierte Analyse gestellt. Dabei zielen die Anforderungen sowohl auf den Schritt der Artefaktdatenanalyse als auch auf die anderen Schritte (siehe Kapitel 7) ab.

TRQ-1 Aufwand: Der Aufwand zur Erstellung und Anpassung neuer Analysewerkzeuge soll möglichst gering sein.

TRQ-2 Wiederverwendung: Werkzeuge sollen so konzipiert sein, dass sie sich in möglichst vielen Anwendungsfällen wiederverwenden lassen.

TRQ-3 Nutzung: Die Nutzung der Werkzeuge soll flexibel möglich sein. Die folgenden Nutzungsszenarien sollen unterstützt werden:

- **TRQ-3.1 Manuelle Nutzung:** Die Nutzung von Werkzeugen zur Durchführung eines oder mehrere Teilschritte der artefaktbasierten Analyse soll manuell und möglichst komfortabel sowie modular über die Kommandozeile ausführbar sein.
- **TRQ-3.2 Skriptbasierte Werkzeugketten:** Die Kommandozeilenwerkzeuge sollen sich gut per Skript zu einer Werkzeugkette verknüpfen lassen, d.h., einheitlich Parameter, Rückgabewerte und eine einheitliche Fehlerbehandlung werden gefordert.
- **TRQ-3.3 Programmatische Nutzung:** Werkzeuge sollen sich programmatisch zu komplexen Analysewerkzeugen kombinieren lassen.

TRQ-4 Artefakdatenanalyse: Die Artefaktdatenanalyse soll durch Werkzeuge unterstützt werden:

- **TRQ-4.1 Manuelle Datenexploration:** Es sollen Werkzeuge zum komfortablen manuellen Explorieren von Artefaktdaten erstellt werden.
- **TRQ-4.2 Automatisierte Durchführung:** Es soll die Möglichkeit geben, spezifizierte Artefaktdatenanalysen automatisiert durchzuführen.
- **TRQ-4.3 Verfügbarkeit:** Aktuelle Artefaktdaten und Analyseergebnisse zu spezifizierten Analysen sollen in regelmäßigen Intervallen oder bei Projektänderungen automatisch berechnet und zugänglich gemacht werden.

Kapitel 5.

Modellierungstechnik für die Artefaktmodellierung

In diesem Kapitel wird eine Modellierungstechnik für die Artefaktmodellierung vorgestellt. Die Artefaktmodellierung umfasst alle manuellen und automatisierten Modellierungstätigkeiten, die bei der Durchführung der artefaktbasierten Analyse anfallen oder für diese notwendig sind (siehe Abschnitt 4.4). Zur Modellerstellung werden verschiedene Sprachen der UML/P (siehe Abschnitt 2.4.1) genutzt. Diese Sprachen, deren Verwendung für die Artefaktmodellierung sowie deren Komposition werden in Abschnitt 5.1 vorgestellt. Anschließend wird in Abschnitt 5.2 ein bei der Erstellung projektspezifischer AMe wiederverwendbarer AM-Kern als Teil der Modellierungstechnik spezifiziert.

5.1. Verwendung der UML/P zur Modellierung von Artefaktzusammenhängen

Die UML/P (siehe Abschnitt 2.4.1) ist ein Sprachprofil der UML. Bei dem in dieser Arbeit verwendeten Ansatz zur Artefaktmodellierung wird eine auf der UML/P basierende Modellierungstechnik verwendet. Die Sprachen der UML sind vielseitig einsetzbar und eignen sich dadurch auch für die Artefaktmodellierung. Darüber hinaus existieren am Lehrstuhl für Softwareengineering bereits Implementierungen für diese Sprachen und auch Werkzeuge, die bei der Erstellung und Weiterverarbeitung von Modellen der UML/P helfen. Somit kann an dieser Stelle auf die besehenden Sprachen und deren Implementierungen zurückgegriffen werden, wodurch Anforderung *MRQ-1* erfüllt wird.

Die vielfältigen Einsatzmöglichkeiten der UML/P zeigen sich beispielsweise bei Klassendiagrammen, die sowohl als konzeptuelle Diagramme des Problemraums oder aber als technische Diagramme des Lösungsraums verwendet werden können. Weiterhin lässt sich ein SC zur Modellierung der möglichen Objektzustände oder aber zur Modellierung von Methodenabläufen verwenden [Rum17]. Je nach Anwendung können sich auch verschiedene Verknüpfungsarten zwischen den Modellen der verschiedenen Sprachen ergeben. Zusätzlich können die Sprachen bei der Wiederverwendung in MDD-Werkzeugen durch das Hinzufügen von CoCos weiter

eingeschränkt werden, sodass solche Werkzeuge nicht alle Sprachkonstrukte der Ursprungssprachen verarbeiten können müssen.

Aus diesen Gründen wird in diesem Abschnitt definiert,

- welche Sprachen verwendet werden.
- für welche Zwecke die Sprachen verwendet werden.
- welche zusätzlichen Einschränkungen für die Sprachen gelten.

Bei der Artefaktmodellierung werden *Klassendiagramme*, *Objektdiagramme* und die *OCL* verwendet. Abbildung 5.1 gibt eine Übersicht über die Verwendungszecke der Modellierungssprachen.

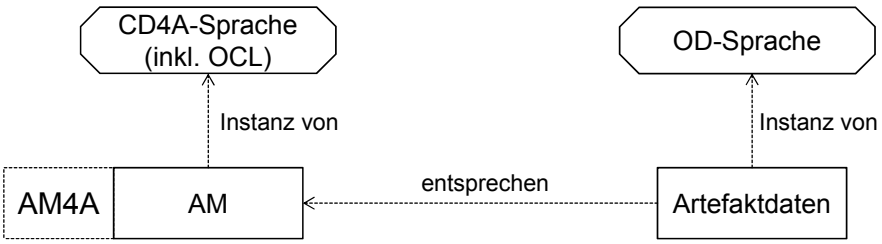

Abbildung 5.1.: Übersicht über den Zusammenhang der verwendeten Sprachen und Modelle bei der Artefaktmodellierung.

In Abbildung 5.1 wird zwischen zwei Arten von Beziehungen unterschieden. Nach [AK03] handelt es sich bei der *Instanz-von*-Beziehung um linguistische Instanziierung, durch welche angegeben wird, dass Modelle in der entsprechenden Sprache definiert wurden. Die *entsprechen*-Beziehung stellt hingegen eine ontologische Instanziierung nach [AK03] dar, bei der das eine Modell eine Instanz des anderen Modells darstellt.

AMe werden, genau wie deren Erweiterungen zu Analysezwecken (AM4A), mit Hilfe von CDs und der OCL modelliert. AMe bilden eine Instanz der zugehörigen Sprache. Diese Sprache ist eine um die OCL erweiterte Variante der CD4A, d.h., innerhalb des textuell definierten Klassendiagramms können ebenfalls OCL-Ausdrücke angegeben werden. Für Analysezwecke lässt sich ein AM erweitern. Diese Erweiterungen sind für sich alleine gesehen keine Instanz der Sprache, da sie auf Modellelemente eines AMs zurückgreifen. Nur zusammen mit dem AM bilden sie gültige Sprachinstanzen. Artefaktdaten werden hingegen durch ODs dargestellt. Die Artefaktdaten entsprechen dabei einem AM (mit oder ohne Erweiterung zu Analysenzwecken).

Im Folgenden wird nun im Detail auf die einzelnen Modellierungssprachen sowie deren Verwendung und Komposition bei der Artefaktmodellierung eingegangen.

5.1.1. Klassendiagramme

Die Sprache CD4A [Rum16, Rot17] wird in dieser Arbeit dazu verwendet, die Struktur von AMen zu modellieren. Dabei werden alle relevanten Elemente des zu analysierenden MDD-

Projekts als Klassen im CD und Beziehungen zwischen Elementen als Assoziationen oder Kompositionen modelliert. Auf die Verwendung von Aggregationen wird hingegen verzichtet, da es für deren Bedeutung vielfältige Interpretationsmöglichkeiten gibt [Rum11]. Zusätzliche Bedingungen eines AMs können mit Hilfe der OCL modelliert werden. Als Datentypen für Eigenschaften der Elemente können primitive Datentypen, `String` und Typen aus dem entsprechenden Modell genutzt werden. Im Gegensatz zur ursprünglichen Sprachdefinition aus [Rot17] kann außerdem `Date` als vordefinierter Datentyp verwendet werden. Als einzige Modifier werden `derived` und `abstract` verwendet. Mit `derived` wird eine Eigenschaft als abgeleitete Eigenschaft gekennzeichnet.

Definition 5.1.1 (Abgeleitete Eigenschaft)**.** *Eine abgeleitete Eigenschaft eines CDs ist eine abgeleitete Assoziation, abgeleitete Komposition oder ein abgeleitetes Attribut. Die Werte solcher Eigenschaften lassen sich aus anderen Eigenschaften berechnen.*

Mit Hilfe von `abstract` lassen sich hingegen abstrakte Klassen im AM modellieren. Diese können dazu genutzt werden Eigenschaften von Elementen im Modell zusammenzufassen. Instanzen dieser Elemente gehören dann immer auch einer spezialisierten Klasse des AMs an. Weitere Modifier werden von der CD4A nicht unterstützt, sind hier aber auch nicht notwendig, da es sich bei einem AM um ein konzeptionelles Modell der Problemdomäne handelt. Für die Modellierung von Beziehungen werden in dieser Arbeit nur unidirektionale Beziehungen verwendet, da sich an dieser Stelle für Beziehungen zwischen Elementen an der Definition des Tracebegriffs aus Abschnitt 3.2 orientiert wurde.

Bei der Modellierung von AMen kann auf einen wiederverwendbaren AM-Kern (siehe Abschnitt 5.2) zurückgegriffen werden. Eine Methodik zur Modellierung von AMen wird in Abschnitt 7.1 beschrieben. Neben der Modellierung von AMen wird die CD4A zusammen mit der OCL auch zur Spezifikation von Analysen verwendet. Wie in Abschnitt 7.2 beschrieben, werden dazu Erweiterungen des AMs zu Analysezwecken erstellt.

5.1.2. Objektdiagramme

Die Sprache der ODs [Rum16] wird in dieser Arbeit dazu genutzt, die Artefaktdaten darzustellen, wodurch Artefaktdaten gut für Menschen lesbar sind (Anforderung *MRQ-5*) und maschinell verarbeitet werden können (Anforderung *MRQ-6*). Objekte repräsentieren dabei konkrete Artefakte, Werkzeuge und weitere Elemente des untersuchten MDD-Projekts, die sich zu dem gewählten Zeitpunkt in dem MDD-Projekt wiederfinden. Darüber hinaus können auch Aktionen, die bei Ablauf der Werkzeugkette bis zu dem gewählten Zeitpunkt bereits durchgeführt wurden in den Artefaktdaten enthalten sein (vgl. Abschnitt 4.4). Bei den Artefaktdaten wird davon ausgegangen, dass ein zugehöriges AM existiert. Objekte werden immer sowohl mit einem `Objektidentifikator` als auch mit dem konkreten `Objekttyp` angegeben. Somit lassen sich Objekte eindeutig einer Klasse des Artefaktmodells zuordnen. Aus diesem Grund müssen keine Attributtypen, sondern lediglich Name und Wert des Attributs angegeben

werden. Die zugehörigen Attributtypen lassen sich durch den Attributnamen und die zugehörige Klasse aus dem Artefaktmodell bestimmen.

Links des ODs stellen Beziehungen zwischen Elementen des MDD-Projekts dar. Links werden immer vollständig angegeben, d.h., dass sowohl die Richtung des Links als auch der Linkname angegeben werden. Nur für Beziehungsarten, für die im zugehörigen AM weder Name noch Rollenname definiert wurden, darf der Linkname weggelassen werden. Für die Angabe des Linknamen gelten folgende Regeln. Alle Regeln für Assoziationen gelten auch für Kompositionen.

- Wenn ein Assoziationsname für die Beziehung im Artefaktmodell vorhanden ist, wird dieser als Linkname verwendet.
- Wenn kein Assoziationsname aber ein Rollenname für die Beziehung im Artefaktmodell vorhanden ist, wird der Rollenname als Linkname verwendet.
- Wenn weder Assoziationsname noch Rollenname vorhanden sind, wird ein Link ohne Linknamen angegeben.

In den ersten beiden Fällen lässt sich eine eindeutige Zuordnung zu den Assoziationen im Artefaktmodell finden, da die CD4A fordert, dass im ersten Fall der Assoziationsname und im zweiten Fall der Rollenname von einer Quellklasse zu einer Zielklasse eindeutig sein muss. Im letzten Fall lässt sich eine Zuordnung zu einer Assoziation ebenfalls vornehmen, da unter Berücksichtigung der Klassenhierarchie nur eine Assoziation für ein Paar aus Quell- und Zielklasse ohne Assoziations- und Rollennamen erlaubt ist.

Beziehungen zwischen den Artefakten lassen sich dadurch bestimmen, dass der Inhalt der Artefakte manuell oder automatisiert analysiert wird. Wird beispielsweise eine Referenz auf ein anderes Artefakt gefunden, so lässt sich daraus eine Beziehungen zwischen den beiden Artefakten ableiten. Wird jedoch mehrfach dieselbe Art der Beziehung zwischen den Artefakten gefunden, so wird in den Artefaktdaten trotzdem nur ein einzelner Link dieser Art erstellt. Dies reicht aus, um die Artefaktstruktur abzubilden. An dieser Stelle trägt das Auslassen zusätzlicher Details zur Erfüllung von Anforderung *MRQ-2* bei. Die Informationen über die einzelnen vorhandenen Beziehungen können in Zusatzdateien gespeichert werden (beispielsweise in Reports, siehe Abschnitt 8.1.1).

Werte von abgeleiteten Eigenschaften können, müssen aber nicht, in den Artefaktdaten dargestellt sein (Details dazu werden in Abschnitt 5.1.4 vorgestellt). Dadurch ist es möglich, zunächst die Daten ohne die Berechnung abgeleiteter Eigenschaften zu erheben und in einem weiteren Schritt die abgeleiteten Eigenschaften zu berechnen. Unter der Annahme, dass die weiteren Regeln eingehalten wurden, stellen beide ODs korrekte Artefaktdaten dar.

Die verwendete Version der OD-Sprache unterstützt zusätzlich zu der in [Sch12] beschriebenen Version, hierarchisch verschachtelte Objekte (siehe Abschnitt 2.4.1). Für hierarchisch verschachtelte Objekte gelten die gleichen, oben beschriebenen Einschränkungen wie für

nichtverschachtelte Objekte. Ebenfalls muss der Name des Attributs, das einen inneren Link repräsentiert, entsprechend den oben genannten Regeln für Links angegeben werden.

Artefaktdaten können auf verschiedene Artefakte verteilt vorliegen und müssen somit nicht in einem einzigen OD repräsentiert sein. Zum einen kann so nur ein Teil des MDD-Projekts abgebildet werden und zum anderen ist explizit erlaubt, dass einzelne ODs nur Daten zu bestimmten Teilen des zugehörigen Artefaktmodells repräsentieren. So dürfen Instanzen von Klassen, Attributen oder Assoziationen des AMs weggelassen werden. Die Entscheidung, ob ein Element, eine Eigenschaft oder eine Beziehung in den Artefaktdaten dargestellt wird oder nicht, wird auf Artefaktmodellebene getroffen, d.h., für ein Element, eine Eigenschaft oder eine Beziehung des AMs werden in den Artefaktdaten entweder alle Instanzen oder keine Instanz dargestellt. Es ist nicht möglich, dass ein Element des Artefaktmodells, z.B. ein Attribut einer Klasse, für manche der zugehörigen Objekte dargestellt wird und für andere nicht. Diese Vorgabe schließt auch abgeleitete Eigenschaften mit ein. Durch die Vorgabe, dass nicht alle Eigenschaften in einem einzelnen OD dargestellt werden müssen, lassen sich die Artefaktdaten auf mehrere ODs verteilen. Dies hat den Vorteil, dass für die Erhebung der Artefaktdaten kleine, wiederverwendbare Extraktionswerkzeuge eingesetzt werden können (siehe Kapitel 8). Dadurch lässt sich der Aufwand für die Datenerhebung bei neuen Projekten reduzieren.

5.1.3. Object Constraint Language

Die Einsatzmöglichkeiten der OCL sind vielfältig und sie lässt sich mit mehreren anderen Modellierungssprachen der UML kombinieren [Rum16]. In dieser Arbeit wird die OCL ausschließlich in Kombination mit Klassendiagrammen verwendet. Trotzdem wird sie hier für verschiedene Zwecke eingesetzt, um

- die Struktur der möglichen Artefaktdaten durch die Angabe von OCL-Bedingungen weiter einzuschränken. Dabei werden *Prüfbedingungen* innerhalb des AM-CDs direkt angegeben.

- *Berechnungsvorschriften* für abgeleitete Eigenschaften zu definieren. Dazu muss genau spezifiziert werden, wie sich eine solche Eigenschaft aus anderen Artefaktdaten berechnen lässt. Bei der Berechnung können auch andere abgeleitete Eigenschaften verwendet werden, sofern sich diese ebenfalls eindeutig durch die Artefaktdaten berechnen lassen. Dies bedeutet insbesondere, dass abgeleitete Eigenschaften sich nicht gegenseitig (auch nicht transitiv) zur Berechnung nutzen dürfen. Berechnungsvorschriften werden wie Prüfbedingungen direkt innerhalb des AM-CDs angegeben.

- die Artefaktdaten zu analysieren. Dazu lassen sich Ergänzungen des AMs zu Analysezwecken analog zu den oben beschriebenen Anwendungen spezifizieren (siehe Abschnitt 7.2). Diese werden in dieser Arbeit als *Prüfbedingungen* bzw. *Berechnungsvorschriften zu Analysezwecken* bezeichnet.

Bei der Definition von OCL-Berechnungsvorschriften bietet es sich an, Ausdrücke der folgenden Formen zu verwenden:

$$x == \dots$$
$$x.contains(\dots)$$
$$x.containsAll(\dots)$$

Hierbei bezeichnet x die zu berechnende abgeleitete Eigenschaft. Diese steht im ersten Fall alleinig auf der linken Seite des Vergleichsoperators $==$, während auf der rechten Seite dessen Berechnung spezifiziert wird. Die anderen beiden Fälle eignen sich vor allem für abgeleitete Assoziationen, deren Berechnung über mehrere OCL-Ausdrücke verteilt definiert wird. Hierdurch ist auch eine Spezialisierung von Assoziationen möglich (siehe Abschnitt 7.1). Die Verwendung dieser Formen von OCL-Ausdrücken ermöglicht es die Berechnungsvorschriften systematisch und sogar automatisiert (siehe Abschnitt 8.4.3) weiterzuverarbeiten.

5.1.4. Semantik

Die Semantik einer Sprache wird nach [HR04] durch die semantische Domäne und eine Abbildung von der Syntax der Sprache in die semantische Domäne definiert. In diesem Abschnitt wird auf die Semantik der verwendeten Sprachen für die Artefaktmodellierung in Ergänzung zu den bereits vorgestellten Sprachcharakteristika eingegangen. Im Rahmen dieser Arbeit wird die semantische Domäne ähnlich zu [MRR11b] als *Object Model* (OM) und somit als endliche Menge aller Objektstrukturen, definiert. Die geforderten syntaktischen Charakteristika und die jeweiligen Abbildungen in die semantische Domäne für die Artefaktmodellierung werden im Folgenden informell beschrieben.

Der Artefaktmodellierung liegt für einen bestimmten Artefaktdatensatz entweder eine *Closed World Assumption* oder eine *Open World Assumption* zugrunde. Ersteres bedeutet, dass alle relevanten Konzepte und deren Beziehungen vollständig modelliert werden und fehlende Elemente als nicht existent zu interpretieren sind. In [Rum17] können dazu Diagramme mit Hilfe des Stereotypen «`complete`» gekennzeichnet werden, wodurch explizit dargestellt wird, dass die Closed World Assumption gilt. Diese Annahme hat den Vorteil, dass sich eindeutig und effizient bestimmen lässt, ob Artefaktdaten zu einem AM konform sind. Dazu muss das verwendete AM jeweils soweit auf die Projektsituation zugeschnitten sein, dass die Existenz von zusätzlichen, nicht in den Artefaktdaten vorgesehenen Objekten oder Attributen nicht gefordert wird. Assoziationen, deren Quell- oder Zielkardinalität 1 ist, fordern beispielsweise, dass genau ein Objekt verlinkt wird. Fehlt eine entsprechende Verlinkung, so liegt die durch das OD repräsentierte Objektstruktur nicht in der Semantik des Klassendiagramms. Auch durch OCL-Bedingungen kann die Existenz bestimmter Objekte oder Links gefordert werden. Unterspezifikation ist somit unter der beschriebenen Annahme nicht erlaubt.

Da eine solche Closed World Assumption nicht immer von Vorteil ist, kann diese strikte Annahme bei Bedarf abgeschwächt werden. Dadurch werden Prüfungen zwischen AMen und

Artefaktdaten auch dann möglich, wenn die Artefaktdaten nicht in vollständiger Form vorliegen oder ein AM verwendet wird, dass die Existenz zusätzlicher Objekte fordert. In diesen Fällen kann für Artefaktdaten eine eingeschränkte Form einer Open World Assumption angenommen werden. Für nicht dargestellte Objekte und Links kann so deren Existenz dennoch angenommen werden. Trotzdem gelten hier dieselben Einschränkungen wie für abgeleitete Eigenschaften: Entweder alle Elemente/Links/Attribute einer zugehörigen Klasse/Assoziation sind dargestellt oder keines. Dadurch kann eine Prüfung zwischen AM und Artefaktdaten weiterhin effizient durchgeführt werden.

Durch die Verwendung dieser Open World Assumption lassen sich Artefaktdaten in separaten ODs speichern, die erst durch ein Zusammenfügen (siehe Abschnitt 7.4) die Konformität zu dem AM bei Verwendung der Closed World Assumption erreichen. Dadurch ist es möglich bei der automatisierten Extraktion von Artefaktdaten auf kleine, wiederverwendbare Werkzeuge zurückzugreifen und trotzdem die Struktur der Artefaktdaten direkt nach der Extraktion prüfen zu können. Außerdem kann die Verwendung der Open World Assumption auch dann sinnvoll sein, wenn das verwendete AM mehr Inhalte fordert als derzeit in den Artefaktdaten bereitgestellt werden. Eine solche Situation kann eintreten, wenn ein AM ohne projektspezifisches Zuschneiden wiederverwendet werden soll oder durch eine iterative Vorgehensweise bei der artefaktbasierten Analyse noch nicht alle Daten erhoben werden können.

Die Prüfung auf Konsistenz zwischen AM und Artefaktdaten lässt sich in zwei Teilprüfungen aufteilen. Zum einen lässt sich prüfen, ob die Artefaktdaten zu dem Klassendiagrammanteil des AMs passen, zum anderen können die OCL-Prüfbedingungen ausgewertet werden. Während für die Auswertung von OCL keine weiteren Spezialisierungen gegenüber [Rum16, Rum17] notwendig sind, wurde für die Konsistenzprüfung zwischen dem Klassendiagrammanteil des AMs und den Artefaktdaten ein eigener Algorithmus entwickelt. Im Folgenden werden zunächst einige Definitionen angegeben bevor der Algorithmus vorgestellt wird.

Definition 5.1.2 (am_{CD})**.** *Der CD-Anteil eines AMs ist am_{CD}, was dem Teil des AMs entspricht, der bei Wegnahme der spezifizierten OCL-Bedingungen übrig bleibt. Ein am_{CD} stellt somit ein gültiges CD dar.*

Definition 5.1.3 (Strukturprüfung)**.** *Die Strukturprüfung prüft, ob ein am_{CD} konsistent zu den Artefaktdaten ist.*

Da die Artefaktdaten in Form eines ODs vorliegen, ist die Strukturprüfung somit mit der Konsistenzprüfung zwischen CDs und ODs gleichzusetzen. Dabei lassen sich die hier vorausgesetzten Charakteristika für CDs und ODs den Features des Feature Diagramms aus [MRR11b] zuordnen. Für CDs muss demnach gelten:

- Alle Attribute sind dargestellt.
- Alle Klassen sind dargestellt.
- Das leere OM ist invalide.

Für ODs werden folgende Charakteristika bei Verwendung der Closed World Assumption vorausgesetzt:

- Alle Objekte sind dargestellt.
- Alle nichtabgeleiteten Attribute sind dargestellt.
- Alle Links sind dargestellt.
- Alle Typen sind dargestellt.
- Es wird eine strikte Objekttypisierung verwendet.
- Das leere OM ist invalide.

Zu beachten ist an dieser Stelle, dass „*Alle nichtabgeleiteten Attribute sind dargestellt*“ kein Feature des Feature Diagramms aus [MRR11b] darstellt. An dieser Stellt wurde das Feature Diagramm um das entsprechende Feature ergänzt. Als zusätzliche Charakteristik wird vorausgesetzt, dass eine einzelne abgeleitete Eigenschaft des CDs in einem zugehörigen OD entweder vollständig für alle Objekte oder für keins der Objekte angegeben ist.

Bei der Verwendung einer Open World Assumption wird die Vollständigkeit der Artefaktdaten nicht angenommen, d.h., die Features „*Alle Objekte sind dargestellt*“, „*Alle nichtabgeleiteten Attribute sind dargestellt*“ und „*Alle Links sind dargestellt*“ aus dem Feature Diagramm aus [MRR11b] gelten dann nicht. Analog zur Angabe von abgeleiteten Eigenschaften gilt hier, dass entweder alle Objekte einer Klasse oder keins dieser Objekte dargestellt sind und, dass eine einzelne Eigenschaft des CDs entweder vollständig für alle Objekte oder für keins der Objekte im OD angegebene ist.

Durch die geforderten Charakteristika ist eine effiziente Konsistenzprüfung zwischen CDs und ODs möglich. Der Algorithmus für die Strukturprüfung bei Verwendung der Closed World Assumption ist wie folgt definiert und ist eine Weiterentwicklung des Algorithmus aus [Hil16]. Angegebene Eigenschaften (abgeleitete und nichtabgeleitete) werden im Folgenden als geforderte Eigenschaften bezeichnet. Bei der Verwendung der Open World Assumption müssen die Schritte (2b) und (4) ausgelassen werden.

1. Prüfe für jedes Objekt im OD, ob ein entsprechender Typ im CD definiert wurde. Diese Prüfung ist einfach möglich, da eine strikte Typisierung der Objekte gefordert ist und alle Klassen im CD dargestellt sein müssen.
2. Prüfe für jedes Objekt im OD, dass
   a) für jedes angegebene Attribut ein entsprechendes Attribut des zugeordneten Typs im CD existiert. Ein entsprechendes Attribut muss sowohl denselben Namen als auch einen zu dem angegebenen Wert passenden Datentyp haben. Diese Prüfung ist möglich, da im CD alle Attribute vollständig angegeben sein müssen, was sowohl den Namen als auch den Typ mit einschließt.

b) alle geforderten Eigenschaften der zugehörigen Klasse des CDs angegeben wurden.

3. Prüfe für jeden Link des ODs, dass genau eine Assoziation im CD existiert, der der Link zugeordnet werden kann. Diese Prüfung ist möglich, da die Zuordnung des Quellobjekts eines Links zu dem entsprechenden Typ im CD wie in (1) beschrieben möglich ist und alle ausgehenden Assoziationen der zugehörigen Typhierarchie eindeutig referenzierbar sind. Die Verwendung der eindeutigen Referenz als Linknamen ist für Links des OD verpflichtend.

4. Prüfe für jede Assoziation aus den geforderten Eigenschaften, ob die Kardinalitätsrestriktionen erfüllt sind.

5.2. Artefaktmodellkern

In diesem Abschnitt wird der Artefaktmodellkern vorgestellt. Dieser spezifiziert auf abstrakter Ebene einige der grundlegenden Konzepte für die Artefaktmodellierung, welche bereits in [GHR17] veröffentlicht wurden. Um die abgeleiteten Beziehungen des AM-Kerns berechnen zu können, muss er durch projektspezifische Konzepte ergänzt werden. Für MontiCore-basierte MDD-Projekte wird dies in Kapitel 6 vorgenommen.

Der in diesem Kapitel beschriebene AM-Kern und später auch das projektspezifische AM für MontiCore-basierte MDD-Projekte werden in Form eines textuellen Modells mit der in Abschnitt 5.1 vorgestellten Modellierungstechnik unter Verwendung der Sprachen CD und OCL definiert. Darüber hinaus werden graphische CDs genutzt, um einen Überblick über die textuell modellierten Sachverhalte zu geben. Die graphischen CDs stellen dabei ein Teil des textuell modellierten AMs dar. Attribute und OCL-Bedingungen werden aus Gründen der Übersichtlichkeit nicht graphisch dargestellt. Sowohl der AM-Kern als auch das in Kapitel 6 vorgestellte projektspezifische AM werden inkrementell vorgestellt, d.h. es werden jeweils einzelne Teile vorgestellt, die zusammen ein integriertes AM bilden. Dabei wird mit der Beschreibung der abstraktesten Elemente und Beziehungen begonnen. Der in diesem Abschnitt vorgestellte AM-Kern dient als Ausgangspunkt für projektspezifische AMe und erfüllt somit Anforderung *MRQ-3.1*. Eine Methodik zur Erstellung von projektspezifischen AMen unter Verwendung des AM-Kerns wird später in Abschnitt 7.1 vorgestellt.

5.2.1. Artefakte und Artefaktcontainer

Die zentralen Elemente jedes AMs sind die Artefakte. Dabei wird die Definition aus Abschnitt 4.1 für den Artefaktbegriff verwendet. Somit können alle projektspezifischen Dateien und Ordner als Artefakte angesehen werden.

Artefakte können sich gegenseitig beinhalten. Typische Beispiele für Artefakte, die andere Artefakte beinhalten können, sind Archive oder Ordner des Dateisystems, aber auch Datenbankdateien sind möglich. Diese Sichtweise wird auch in [Bal11] eingenommen. Dort wird

als Beispiel für ineinander geschachtelte Artefakte ebenfalls ein Jar-Archiv mit beinhaltenden .class-Dateien gezeigt. In Abbildung 5.2 ist ein Überblick über die physische Organisation von Artefakten dargestellt.

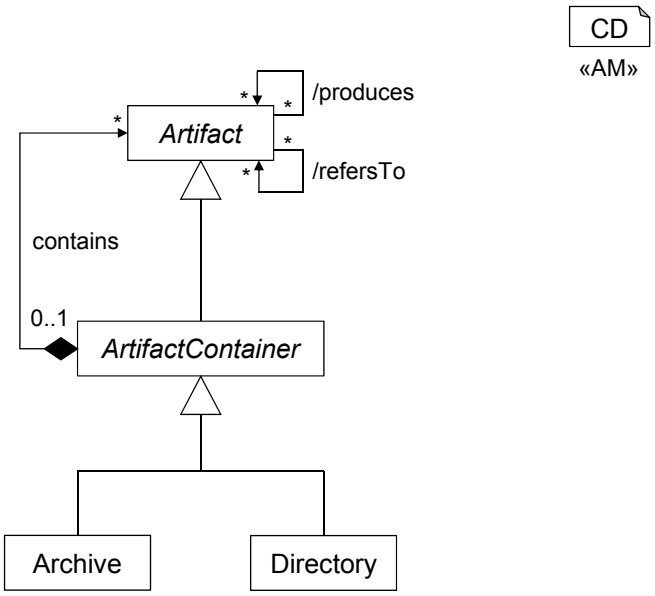

Abbildung 5.2.: Abstrakte Beziehungen zwischen Artefakten und die Organisation von Artefakten in Containern.

Da für diesen Teil des AM-Kerns das Composite Pattern [GHJV95] genutzt wurde, können Archive und Ordner sich gegenseitig in beliebiger Reihenfolge enthalten. Dabei ist jedes Artefakt höchstens in einem Artefaktcontainer enthalten. Wenn alle verfügbaren Artefakte modelliert sind, existiert genau ein Artefakt, das nicht in einem Container enthalten ist: Das Wurzelverzeichnis des Dateisystems (siehe Listing 5.6). Darüber hinaus können Artefakte dazu genutzt werden andere Artefakte zu erstellen (`produce`-Beziehung) und sie können andere Artefakte statisch referenzieren (`refersTo`-Beziehung). Diese Beziehungen werden später in Listing 5.4 genauer spezifiziert.

Artefakte Artefakte sind die zentralen Elemente von AMen. Es existieren dabei verschiedenen Artefaktarten die sich von Projekt zu Projekt unterscheiden können. Ihre gemeinsamen Eigenschaften werden durch die in Listing 5.3 dargestellte abstrakte Klasse `Artifact` modelliert. Jedes Artefakt hat einen Kurznamen (`simpleName`) und eine Namenserweiterung `nameExtension`. Diese bilden zusammen einen eindeutigen Namen innerhalb eines Containers, wodurch sich Artefakte eindeutig referenzieren lassen. Zusätzlich lässt sich auf das Datum der letzten Änderung (`modified`) zurückgreifen und es wurden eine Reihe von abgeleiteten Attributen modelliert, von denen in Listing 5.3 nur das `name`-Attribut durch eine OCL-Berechnungsvorschrift definiert wird. Im Fall, dass die Namenserweiterung leer ist, gleicht der Name dem Kurznamen. In allen anderen Fällen wird der Name aus Kurzname

```
1  abstract class Artifact {                                    CD
2    String simpleName;                                         «AM»
3    String nameExtension;
4    Date modified;
5    /String name;
6    /String fullName;
7    /boolean isRoot;
8  }
9
10 context Artifact inv:
11   name == ((nameExtension == "") ? simpleName :
12     simpleName + "." + nameExtension);
```

Listing 5.3: AM: `Artifact`.

und Namenserweiterung, getrennt durch einen Punkt, berechnet. Die Berechnung der anderen abgeleiteten Attribute wird später spezifiziert.

```
1  association /refersTo [*] Artifact -> Artifact [*];          CD
2                                                               «AM»
3  association /produces [*] Artifact -> Artifact [*];
4
5  context Artifact a inv:
6    !a.produces**.contains(a);
```

Listing 5.4: AM: Fundamentale Beziehungen.

Beziehungen zwischen Artefakten In Abschnitt 3.2 wurden drei Arten von fundamentalen Beziehungen zwischen Artefakten vorgestellt. Die *bezieht-sich-auf*-Beziehung (`refersTo`) und die *produziert*-Beziehung (`produces`) sind in Listing 5.4 dargestellt. Die *beinhaltet*-Beziehung (`contains`) folgt in in Listing 5.5.

Die erste der zentralen Artefaktbeziehungen des AM-Kerns ist die `refersTo`-Beziehung. Speziellere Formen dieser Beziehungsart werden in der `refersTo`-Beziehung vereinigt. Die Beziehung ist wie folgt definiert.

Definition 5.2.1 (`refersTo`-Beziehung zwischen Artefakten)**.** *Wenn ein Artefakt Informationen von einem anderen Artefakt benötigt, um seinen Zweck zu erfüllen, dann bezieht es sich (`refersTo`) auf das andere Artefakt.*

Ein Beispiel für eine spezifische `refersTo`-Beziehung ist die Beziehung zwischen Java-Dateien die einander benötigen, da Typen und Methoden benötigt werden, die in dem anderen

Artefakt definiert wurden. Innerhalb von Java-Dateien können solche Beziehungen durch `import`-Statements oder die Nutzung qualifizierter Typnamen hervorgerufen werden. Normalerweise werden `refersTo`-Beziehungen durch die Referenzierung von Namen hervorgerufen. Es können dabei zyklische Artefaktbeziehungen auftreten.

Die `produces`-Beziehung ist eine weitere zentrale Beziehung des AM-Kerns. Sie beschreibt, dass Informationen aus einem Artefakt für die Erstellung eines anderen Artefakts genutzt wurden. Ein Artefakt kann automatisch durch die Ausführung von Werkzeugen erstellt werden, wobei existierende Artefakte als Eingabe für das Werkzeug dienen. Dies führt zu der folgenden Definition.

Definition 5.2.2 (`produces`-Beziehung zwischen Artefakten)**.** *Ein existierendes Artefakt trägt zur Erstellung des neuen Artefaktes (zu seiner Produktion) bei, wenn seine Existenz und/oder sein Inhalt Einfluss auf das resultierende Artefakt haben.*

Die Erstellung des neuen Artefakts hat keinen Einfluss auf das Quellartefakt, d.h., das Quellartefakt wird dabei weder modifiziert noch gelöscht. Bei der Erstellung werden alle benötigten Informationen der Quellartefakte in die neu erstellten Artefakte überführt. Daraus folgt: (1) Ein resultierendes Artefakt hat normalerweise keine `refersTo`-Beziehung zu seinen Quellen und (2) die `produces`-Beziehungen bilden keine Zyklen. Dies ist auch durch die OCL-Prüfbedingung in Zeile 5 von Listing 5.4 unter Verwendung der transitiven Hülle (`**`) spezifiziert. Es ist aber möglich, dass ein Artefakt aus mehreren Quellen erzeugt wurde oder, dass ein Eingabeartefakt mehrfach zur Erstellung anderer Artefakte genutzt wird.

Werkzeuge bestehen ebenfalls aus einer Menge von Artefakten. Diese Artefakte werden als produzierende Elemente für automatisch erstellte Artefakte angesehen. Das ist vor allem deshalb interessant, da Werkzeuge in dieser Arbeit nicht als unveränderbare Blackboxen angesehen werden, sondern als agil weiterentwickelbare Einheiten (vgl. Abschnitt 2.2). Darüber hinaus lassen sich einige Werkzeuge selbst (teilweise) automatisch durch andere Werkzeuge generieren. Um auch solche Sachverhalte analysieren zu können, werden über die `produces`-Beziehung alle notwendigen Artefakte entsprechend verknüpft, was eine durchgängige Analyse aller an einem erstellten Artefakt beteiligten Artefakte erlaubt.

Auch von der `produces`-Beziehung existieren verschiedene spezifische Ausprägungen. Beispiele, bei denen es zu solchen Beziehungen kommt, sind ein einfaches Kopierskript, Compiler oder Codegeneratoren. Dabei haben alle Beispiele gemeinsam, dass sie automatisch und wiederholbar ausführbar sind und ohne zusätzliche menschliche Interaktion auskommen.

Artefaktcontainer

Definition 5.2.3 (Artefaktcontainer)**.** *Ein Artefaktcontainer ist ein Artefakt, das andere Artefakte enthalten kann.*

Bei der Modellierung von Artefaktcontainern wurde das Composite Pattern [GHJV95] verwendet. Somit ist `ArtifactContainer` eine Spezialisierung von `Artifact`. Artefaktcontainer werden an dieser Stelle ebenfalls als individuell wiederverwendbare Einheiten des

Entwicklungsprojekts angesehen, weshalb diese Art der Modellierung gewählt wurde. Artefaktcontainer erben die Eigenschaften des `Artifact`-Elements, haben aber keine weiteren spezifischen Attribute. Wie durch die `contains`-Beziehung in Abbildung 5.2 dargestellt ist, können Artefaktcontainer beliebige Artefakte enthalten. Im AM-Kern wird zwischen zwei Spezialformen von Artefaktcontainern unterschieden: Ordner und Archive.

CD
«AM»

```
1  abstract class ArtifactContainer extends Artifact {}
2
3  composition contains
4    [0..1] ArtifactContainer (parent) -> Artifact [*];
5
6  context Artifact inv:
7    (isRoot <=> parent.isAbsent) &&
8    (!isRoot implies {p in parent** | p.isRoot}.size == 1);
9
10 context Artifact inv:
11   fullName == (isRoot ? "/" :
12                (parent.isRoot ? "/" + name :
13                 parent.fullName + "/" + name));
14
15 context Artifact a, Artifact b inv:
16   (a.name == b.name && a.parent == b.parent)
17     implies a == b;
```

Listing 5.5: AM: `ArtifactContainer`.

In Listing 5.5 sind weitere OCL-Ausdrücke für Artefaktcontainer angegeben. Der erste Ausdruck stellt sicher, dass das Wurzelverzeichnis das einzige Artefakt ist, das nicht in einem Artefaktcontainer (`parent`) enthalten ist und, dass nur ein Wurzelverzeichnis für jedes modellierte Dateisystem vorhanden ist. Der zweite Ausdruck definiert die Berechnung des `fullName`-Attributs von Artefakten. Dazu wird die Hierarchiestruktur bestehend aus Artefaktcontainern betrachtet. Über das `fullName`-Attribut lassen sich Artefakte eindeutig innerhalb eines Dateisystems referenzieren. In dieser Hierarchiestruktur können sowohl Ordner als auch Archive enthalten sein. Der dritte Ausdruck aus Listing 5.5 stellt sicher, dass das `name`-Attribute von Artefakten innerhalb eines Artefaktcontainers eindeutig ist, was die Eindeutigkeit von `fullName`-Attributen innerhalb eines Dateisystems garantiert.

Die aktuelle Form der Modellierung erlaubt es somit auch mehrere Dateisysteme abzubilden, da bei der Definition der Bedingungen darauf geachtet wurde, sowohl mehrere Wurzelverzeichnisse zuzulassen als auch die Eindeutigkeit von qualifizierten Namen nur für einzelne Dateisysteme zu fordern.

Ordner Ordner des Dateisystems werden durch das `Directory`-Element repräsentiert. Da Ordner, wie oben beschrieben, ebenfalls als Artefakte modelliert wurden, erben auch sie das

```
1  class Directory extends ArtifactContainer {}                    CD
2                                                                   «AM»
3  context Directory inv:
4    nameExtension == "";
5
6  context Artifact inv:
7    isRoot implies (this in Directory);
8
9  context Artifact inv:
10   isRoot <=> (simpleName == "/");
11
12 context Artifact inv:
13   name.contains("/") implies isRoot;
```

Listing 5.6: AM: `Directory`.

`nameExtension`-Attribut. Durch die in Zeile 3 von Listing 5.6 dargestellte Prüfbedingung wird aber sichergestellt, dass dieses für `Directories` immer leer ist. Darüber hinaus wird in dem gleichen Listing spezifiziert, dass das Wurzelartefakt immer ein Verzeichnis (Zeile 6) und dessen Name „/“ ist (Zeile 9). In Zeile 12 wird zusätzlich spezifiziert, dass außer dem Wurzelverzeichnis kein anderes Artefakt einen „/“im Namen haben darf, wodurch sichergestellt ist, dass Artefaktnamen zu einem typischen Dateisystem passen.

Archive Archive werden genutzt, um mehrere zusammengehörige Artefakte in einer einzigen Datei zusammenzufassen und um Speicherung, Austausch und Versionierung zu vereinfachen. Zusätzlich kann der Speicherbedarf durch Komprimierung verringert werden, was jedoch im Rahmen dieser Arbeit nicht weiter betrachtet wird. Im Allgemeinen kann auf den Inhalt von Archiven nicht direkt zugegriffen werden. Stattdessen müssen zu diesem Zweck oftmals externe Programme genutzt werden, um die beinhalteten Daten eines Archivs zu extrahieren. Im Rahmen dieser Arbeit sind jedoch vor allem die Inhalte von Archiven und deren Organisation und nicht Kompressions- oder Extraktionsalgorithmen von Interesse. Deshalb wurden Archive so modelliert, als ob direkt und ohne Einschränkung auf deren Inhalte zugegriffen werden könnte. Im AM-Kern ist das Element `Archiv`, wie in Listing 5.7 gezeigt, als Subtyp von `ArtifactContainer` modelliert. Im Gegensatz zu der Modellierung von Ordnern wurden an dieser Stelle keine weiteren Einschränkungen spezifiziert.

```
1  class Archive extends ArtifactContainer {}                      CD
2                                                                   «AM»
```

Listing 5.7: AM: `Archive`.

5.2.2. Systeme und Werkzeuge

Um zu verstehen, wie Systeme in AMen modelliert werden, wird zunächst die folgende Definition betrachtet.

Systeme

Definition 5.2.4 (System)**.** *Ein* System *ist eine Menge von kooperierenden und verbundenen Artefakten, die ausgeführt werden kann, um einen bestimmten Zweck zu erfüllen.*

Der Begriff System wird somit für ausführbare Softwaresysteme verwendet. Während in C-basierten Projekten ein System häufig als einzelne `.exe`-Datei ausgeliefert wird, werden Java-Systeme als Jar-Archive oder als Menge von `.class`-Dateien in einem Ordner bereitgestellt.

Im AM-Kern wird zwischen zwei verschiedenen Arten von Systemen unterschieden: Werkzeuge und Produkte. Werkzeuge werden während des Entwicklungsprozesses verwendet, wohingegen Produkte das Ergebnis des Softwareentwicklungsprozesses darstellen. Die Betrachtung von Werkzeugen im AM ist gerade dann interessant, wenn Werkzeuge in demselben MDD-Projekt modifiziert und weiterentwickelt werden wie das Produkt. Gerade wenn verschiedene Werkzeuge in einem Projekt eingesetzt werden, die sich gemeinsame Artefakte teilen oder zur Erstellung anderer, ebenfalls in dem Projekt genutzter Werkzeuge beitragen (z.B. durch Generierung), wird die Betrachtung von Werkzeugen interessant.

Abbildung 5.8 gibt eine Übersicht über die zu Systemen zugehörigen Elemente.

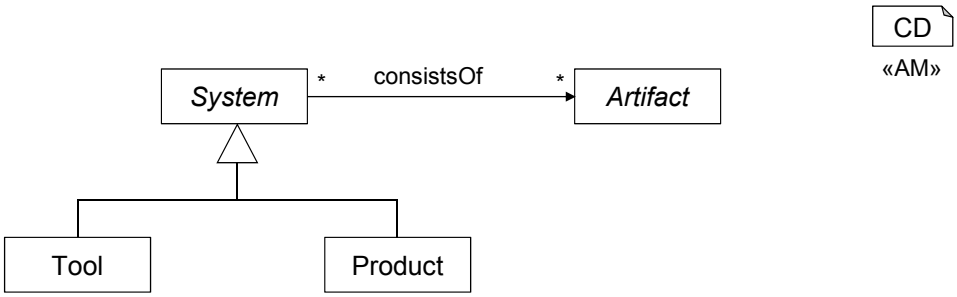

Abbildung 5.8.: Ein System besteht aus Artefakten und ist entweder ein Werkzeug oder ein Produkt.

Systeme lassen sich normalerweise anhand ihrer Namen und ihrer Versionsnummer identifizieren. Dabei hat, wie in Listing 5.9 gezeigt, das `version`-Attribut den Typ String, um beliebige Versionsdefinitionen zuzulassen (z.B. *1.0.0-SNAPSHOT*). Durch die `consistsOf`-Beziehung wird modelliert, dass Werkzeuge selbst aus einer Menge von Artefakten bestehen. Die Art dieser Artefakte wird jedoch nicht weiter beschränkt.

Die beiden vorgestellten Spezialformen von Systemen, Werkzeuge und Produkte, werden in Listing 5.10 und Listing 5.11 definiert.

Ein Produkt stellt dabei das Hauptergebnis des Projekts, welches an die Endnutzer ausgeliefert werden soll, dar. Produkte können dabei durch den Einsatz von Generatoren (teilweise) generiert werden.

```
abstract class System {
  String name;
  String version;
}

association consistsOf [*] System -> Artifact [*];
```

CD «AM»

Listing 5.9: AM: `System`.

```
class Product extends System {}
```

CD «AM»

Listing 5.10: AM: `Product`.

```
class Tool extends System {}
```

CD «AM»

Listing 5.11: AM: `Tool`.

Werkzeuge werden ebenfalls als spezielle Form von Systemen angesehen. Im Gegensatz zu Produkten sind Werkzeuge in den MDD-Buildprozess integriert. Bei dessen Ausführung werden Artefakte durch Werkzeuge eingelesen und von diesen erstellt. Die Modellierung der Werkzeugausführung ist nicht Teil des AM-Kerns, wird aber später in Abschnitt 6.5.3 als Teil des Artefaktmodells für MontiCore-basierte Projekte vorgenommen. Werkzeuge können entweder dem *Entwickler* fertig zur Verfügung stehen oder sie müssen innerhalb des Projekts durch den *Werkzeugentwickler* entwickelt oder angepasst werden. Ein Beispiel für den zweiten Fall ist ein projektspezifischer Codegenerator.

Module Um architektonische Unterstrukturen von Systemen modellieren zu können, werden Systeme im AM-Kern in Module unterteilt. Module bestehen aus Artefakten und lassen sich weiter in Untermodule untergliedern. Abbildung 5.12 zeigt die Modellierung von Modulen. Durch die Verwendung von Modulen lässt sich eine Übersicht über die Systemarchitektur erstellen.

Definition 5.2.5 (Module)**.** *Ein Modul besteht aus einer Menge von kooperierenden und verknüpften Artefakten die gemeinsam einen bestimmten Zweck erfüllen. Module müssen nicht vollständig oder ausführbar sein.*

Diese Definition schließt Module als Subsysteme, wiederverwendbare Frameworks oder Klassenbibliotheken oder auch einzelne Java-Dateien mit ein. Das Paketkonzept von Java kann dazu genutzt werden, Module zu strukturieren, erzwingt dies jedoch nicht. Zusätzlich werden

einzelne Jar-Archive oftmals als Module angesehen. Mit dem Release von Java 9 wurde darüber hinaus ein eigenes Modulkonzept in die Sprache integriert [MB17].

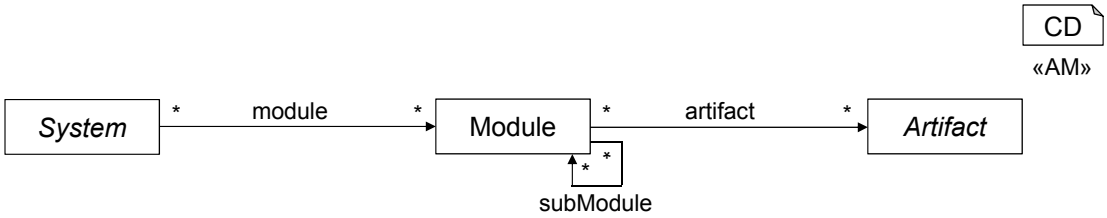

Abbildung 5.12.: Module repräsentieren die architektonische Substruktur des Systems. Sie bestehen aus Artefakten und können sich aus Submodulen zusammensetzen.

Im AM-Kern repräsentieren Module Architektureinheiten, die auf Artefakten und deren Beziehungen basieren. Idealerweise wird die angestrebte Architektur durch die physische Organisation der Artefakte in Artefaktcontainern abgebildet, um eine leichtere Nachvollziehbarkeit zu erlangen. Andere Ansätze, wie beispielsweise [PvdW15], nutzen Typen und deren Beziehungen anstelle von Artefakten, um Systemarchitekturen zu analysieren.

CD
«AM»

```
1  class Module {
2    String name;
3  }
4
5  association module [*] System -> Module [*];
6
7  association subModule [*] Module -> Module [*];
8
9  association artifact [*] Module -> Artifact [*];
```

Listing 5.13: AM: `Module`.

Wie in Listing 5.13 dargestellt, bildet eine Menge von Modulen zusammen ein System. Durch die `subModule`-Beziehung wird die weitere Unterteilung von Modulen in Untermodule ermöglicht, wodurch sich Module hierarchisch verschachteln lassen. Neben einer beliebigen Anzahl von Untermodulen beinhalten Module eine Menge von Artefakten, was durch die `artifact`-Assoziation modelliert ist. Dabei ist nicht ausgeschlossen, dass einzelne Artefakte oder Untermodule verschiedenen Modulen zugeordnet werden. Während Module manchmal als einzelne Artefakte, beispielsweise durch Archive, realisiert sind, kann es ebenfalls sein, dass Module nur in konzeptueller Form bestehen und nicht direkt in der Artefaktstruktur des Projekts abgebildet sind.

Kapitel 6.

Artefaktmodell für modellgetriebene Softwareentwicklungsprojekte mit MontiCore

In diesem Kapitel wird ein AM für Softwareentwicklungsprojekte beschrieben, in denen MontiCore-basierte Werkzeuge eingesetzt werden. Durch den Einsatz dieser Werkzeuge kann ein agiler, modellgetriebener Entwicklungsprozess realisiert werden, welcher einen komplexen Buildprozess beinhalten kann.

Das beschriebene AM basiert auf dem in Abschnitt 4.4 vorgestellten AM-Kern, d.h., die dort gezeigten Klassen und Assoziationen werden in diesem Kapitel wiederverwendet und spezialisiert. Eine Methodik zur systematischen Erstellung von AMen unter Verwendung des AM-Kerns wird in Abschnitt 7.1 vorgestellt.

In diesem Abschnitt werden neben dem AM auch beispielhafte Analysespezifikationen für die artefaktbasierte Analysen vorgestellt (Abschnitt 6.6). Eine zugehörige Methodik zur Erstellung dieser Analysen unter Verwendung eines AMs folgt später in Abschnitt 7.2.

Die Inhalte dieses Kapitels basieren stark auf [GHR17]. Dort werden zusätzlich zu den hier vorgestellten Inhalten u.a. Beziehungen zwischen Java Artefakten anhand von Beispielen, Maven-basierte Buildprozesse und weitere beispielhafte Analysen betrachtet. Das zusammengefügte AM dieses Kapitels inklusive des AM-Kerns ist dieser Arbeit angehängt (Anhang C). Zusätzlich zu den in [GHR17] veröffentlichten Inhalten wird in Abschnitt 6.6.2 auf die Berechnung von Differenzarchitekturen und in Abschnitt 6.7 auf verwandte Modellierungsansätze eingegangen. Außerdem wurden die Inhalte aus [GHR17] an einigen Stellen leicht angepasst und Inkonsistenzen behoben.

6.1. Java Artefakte

Java-Programme werden, wie Programme vieler anderer Programmiersprachen auch, in Dateien gespeichert. Java-Quelldateien und kompilierte .class-Dateien müssen in Java-Projekten organisiert werden. Java nutzt eine smarte Technik, um die Artefakte zu identifizieren, in denen wiederverwendbare Typen definiert wurden. Dies ist beispielsweise bei der Auswertung

von Importstatements der Fall, wo Typen aus anderen Artefakten zur Implementierung des importierenden Artefakts genutzt werden. Im Folgenden werden die Beziehungen zwischen Java-Artefakten untersucht und als Teil des AMs modelliert.

6.1.1. Java-Quellcode- und .class-Dateien

In diesem Abschnitt werden Java-Artefakte zusammen mit ihren verschiedenen Beziehungen untersucht und als Verfeinerung des AM-Kerns (siehe Abschnitt 5.2) modelliert. In Abbildung 6.1 wird eine Übersicht über die modellierten Artefakte aus Java-Softwareprojekten sowie deren Beziehungen geben.

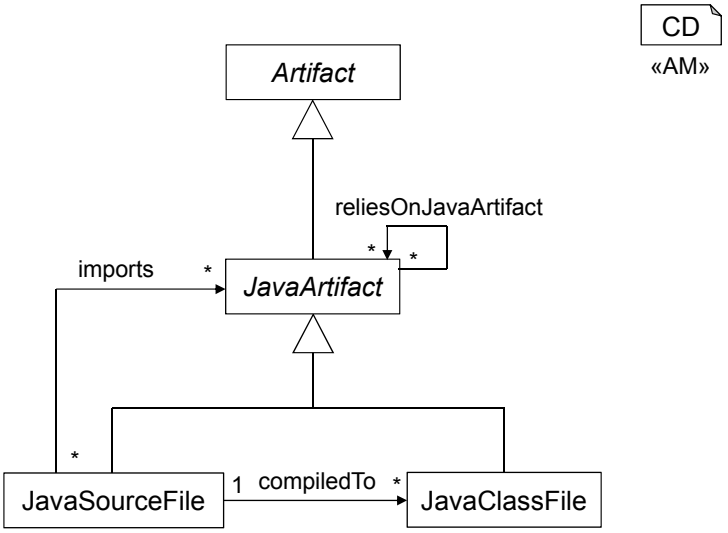

Abbildung 6.1.: Modellierung von Java-Artefakten und zugehörigen Beziehungen.

Ein Java-Artefakt ist entweder eine Java-Quellcodedatei (`JavaSourceFile`) oder eine kompilierte .class-Datei (`JavaClassFile`). Java-Quellcodedateien enthalten in Java [GJS+15] geschriebenen Programmcode, während .class-Dateien kompilierten Programmcode enthalten, der sich in der *Java Virtual Maschine* (JVM) ausführen lässt [LYBB15]. Letztere enthalten ebenfalls die vollständigen Symbolinformationen (vgl. Definition 2.3.2), die für eine externe Verwendung notwendig sind. Deshalb müssen Java-Quellcodedateien nicht zwingend auf andere Quellen verweisen, sondern können ebenfalls Typen aus kompilierten .class Dateien verwenden.

Java-Artefakte Die `JavaArtifact`-Klasse des AMs ist eine Spezialisierung der `Artifact`-Klasse des AM-Kerns. Sie ist abstrakt und vereinheitlicht die gemeinsamen Eigenschaften ihrer beiden Unterklassen `JavaSourceFile` und `JavaClassFile`. Listing 6.2 definiert diese Klassen zusammen mit der zugehörigen Beziehung.

Die reflexive `reliesOnJavaArtifact`-Assoziation aus Abbildung 6.1 gibt an, dass zwei Java Artefakte aufeinander angewiesen sein können. Eine Java-Quellcodedatei ist auf

```
1  abstract class JavaArtifact extends Artifact {}                    CD
2                                                                     «AM»
3  association reliesOnJavaArtifact
4    [*] JavaArtifact -> JavaArtifact [*];
5
6  context JavaArtifact inv:
7    refersTo.containsAll(reliesOnJavaArtifact);
```

Listing 6.2: AM: `JavaArtifact`.

ein anderes Java-Artefakt angewiesen gdw. das andere Artefakt bei der Kompilierung der Quellcodedatei geladen werden muss. Eine .class-Datei ist auf ein anderes Artefakt angewiesen gdw. etwas (Typ, Methode, Konstante, ...) aus der anderen Datei bei der Ausführung des Programmteils, der durch die .class-Datei definiert ist, genutzt wird. Wie durch die OCL-Bedingung aus Listing 6.2 beschrieben wird, ist die `reliesOnJavaArtifact`-Beziehung als Spezialisierung der `refersTo`-Beziehung modelliert.

Java-Quellcodedateien Java-Quellcodedateien sind in Java geschriebene Artefakte. Sie wurden, wie in Listing 6.3 dargestellt, durch die `JavaSourceFile`-Klasse als Spezialisierung der abstrakten `JavaArtifact`-Klasse modelliert.

```
1  class JavaSourceFile extends JavaArtifact {}                       CD
2                                                                     «AM»
3  association imports [*] JavaSourceFile -> JavaArtifact [*];
4
5  association compiledTo
6    [1] JavaSourceFile -> JavaClassFile [*];
7
8  context JavaSourceFile inv:
9    reliesOnJavaArtifact.containsAll(imports);
10
11 context JavaSourceFile inv:
12   produces.containsAll(compiledTo);
13
14 context JavaSourceFile inv:
15   nameExtension == "java";
```

Listing 6.3: AM: `JavaSourceFile`.

Eine Java-Quellcodedatei kann andere Java-Artefakte importieren. Dabei wird ein solcher Import auf Dateiebene durch das Importstatement in der importierenden Java-Datei hervorgerufen. Diese Beziehung wird durch die `imports`-Beziehung des AMs repräsentiert, wobei an dieser

Stelle nicht zwischen Importen von .class-Dateien und Java-Quellcodedateien unterschieden wird. Durch diese Beziehung werden Zugriffe auf die Symbole des importierten Artefakts, wie beispielsweise Typen, Methoden, etc., ermöglicht. Da es sich bei der `imports`-Beziehung um eine Beziehung zwischen zwei Java-Artefakten handelt, trägt diese zur `reliesOnJavaArtifacts`-Beziehung bei (Zeile 8).

Importe können, müssen aber nicht, dazu genutzt werden, um auf Elemente eines anderen Artefakts zuzugreifen. Zum einen lässt sich auf Elemente aus anderen Artefakten auch durch die Verwendung des qualifizierten Elementnamens zugreifen, zum anderen können Importe auch ungenutzt bleiben. Eine gute Richtlinie ist an dieser Stelle, dass möglichst `imports == reliesOnJavaArtifacts` gelten soll. Eine Ausnahme bilden Artefakte, die demselben Paket angehören. Auf deren Elemente kann ohne die Verwendung von Importen oder qualifizierten Namen zugegriffen werden.

Java-Quellcodedateien werden während des Kompiliervorgangs in .class-Dateien übersetzt. Dabei können aus einer einzelnen Quelldatei mehrere kompilierte Dateien entstehen. Dies wird durch die `compiledTo`-Beziehung dargestellt, die selbst eine Verfeinerung der `produces`-Beziehung ist (Zeile 11).

.class-Dateien Nach [LYBB15] ist eine .class-Datei kompilierter Code, der in der JVM ausgeführt werden kann. Als Format wird für .class-Dateien ein hardware- und betriebssystemunabhängiges Binärformat verwendet.

In dem hier vorgestellten AM wird angenommen, dass der kompilierte Code ausschließlich in Dateiform vorliegt. Dies wird durch die `JavaClassFile`-Klasse des AMs modelliert. Die zugehörige Definition wird in Listing 6.4 dargestellt.

CD «AM»

```
1  class JavaClassFile extends JavaArtifact {}
2
3  context JavaClassFile inv:
4    nameExtension == "class";
```

Listing 6.4: AM: `JavaClassFile`.

Analog zur Modellierung von Java-Quellcodedateien stellt die `JavaClassFile`-Klasse ebenfalls eine Spezialisierung der abstrakten `JavaArtifact`-Klasse dar. Somit können beide Artefaktarten durch die gewählte Modellierung der `imports`-Beziehung referenziert werden.

Java-Archive Um die Artefakte aus dem Java-Umfeld zu komplettieren, werden im AM auch Java-Archive (`jar`-Dateien) betrachtet. In Abbildung 6.5 bzw. Listing 6.6 wird die Modellierung von Java-Archiven dargestellt.

Java-Archive dienen dem gleichen Zweck wie andere Archive. Von diesen unterscheiden sie sich dadurch, dass im Wesentlichen Java-Quellcodedateien bzw. .class-Dateien als Inhalt erwartet werden. Trotzdem ist auch hier das Enthaltensein von Artefakten anderer Arten erlaubt.

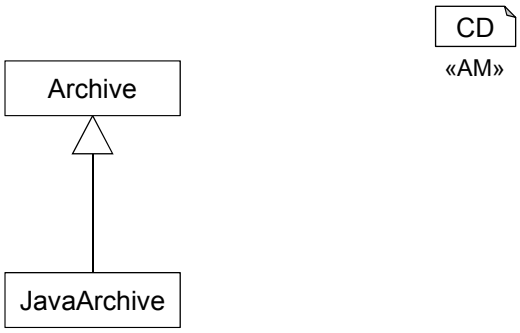

Abbildung 6.5.: Java-Archive.

```
1  class JavaArchive extends Archive {}
2
3  context JavaArchive inv:
4    nameExtension == "jar";
```

CD «AM»

Listing 6.6: AM: `JavaArchive`.

In Java-Archiven wird die gleiche Ordnerstruktur genutzt, wie sie auch außerhalb von Archiven bei Java-Projekten üblich ist. Dies erlaubt Java ein effektives Auffinden der benötigten Artefakte bei der Kompilierung oder Programmausführung [GHR17].

6.1.2. Beziehungen zwischen Java-Artefakten und Java-Typen

Für Java-Projekte zieht das AM neben den bereits vorgestellten Artefakten auch weitere Elemente in Betracht. So werden Java-Typen und Pakete in diesem Kapitel zusammen mit ihren Beziehungen modelliert. Darüber hinaus werden auch die Beziehungen zwischen Java-Artefakten und diesen weiteren Elementen eingeführt. In Abbildung 6.7 wird eine Übersicht der in diesem Kapitel vorgestellten Elemente und Beziehungen gegeben.

Java-Pakete Zunächst wird die `Package`-Klasse des AMs zur Repräsentation von Java-Paketen genauer betrachtet. Pakete organisieren eine Menge zugehöriger Klassen und Interfaces [Ora16].

Die `Package`-Klasse enthält ein `name`-Attribut sowie ein `fullName`-Attribut. Darüber hinaus gehört jedes Java-Artefakt zu genau einem Paket, was durch die `belongsTo`-Beziehung in Abbildung 6.7 dargestellt wird. Die durch das Namensschema von Paketen vorgegebene Pakethierarchie wird durch die `subPackage`-Beziehung modelliert. Diese hierarchische Interpretation zwischen Paketen basiert auf der Spezifikation [GJS+15]. Während eine solche Paketstruktur hilfreich ist, um Programmcode zu strukturieren und somit dessen Verständlichkeit zu erhöhen, hat die `subPackage`-Beziehung für Java selbst keine Bedeutung. So

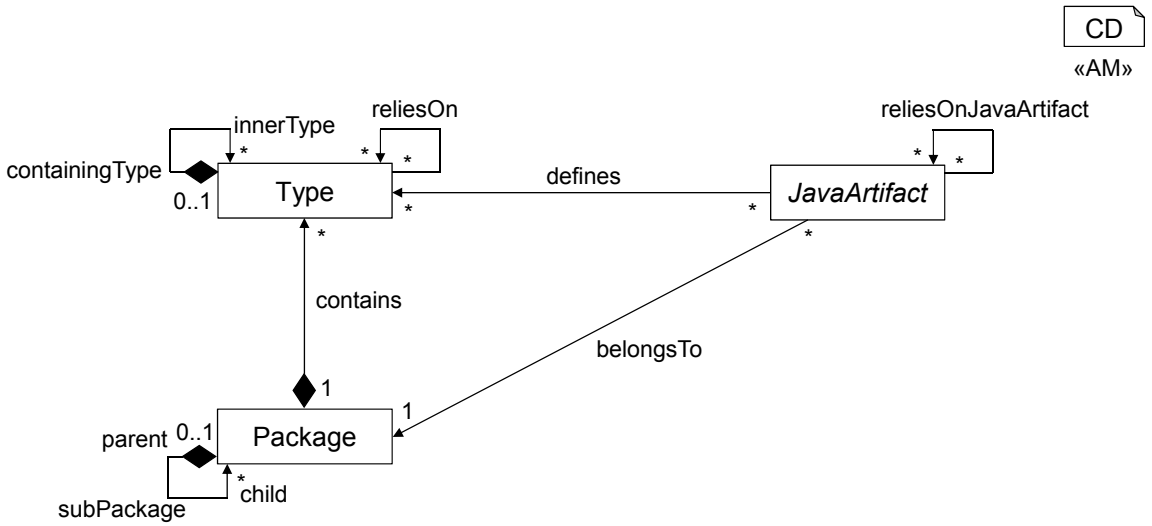

Abbildung 6.7.: Java-Typen und deren Beziehungen.

CD
«AM»

```
class Package {
  String name;
  String fullName;
  /boolean isRoot;
}

association belongsTo [*] JavaArtifact -> Package [1];

composition subPackage
  [0..1] Package (parent) -> (child) Package [*];

context Package inv:
  !name.contains(".");

context Package inv:
  (isRoot <=> parent.isAbsent) &&
  (!isRoot implies {p in parent** | p.isRoot}.size == 1);

context Package inv:
  !isRoot implies
    fullName == parent.fullName + "." + name;

context Package a, Package b inv:
  (a.parent == b.parent && a.name == b.name) implies a == b;
```

Listing 6.8: AM: `Package`.

besteht beispielsweise bei der Sichtbarkeit von Elementen aus Ober- und Unterpaketen kein Unterschied zur Sichtbarkeit von Elementen aus beliebigen anderen Paketen.

In Listing 6.8 werden mehrere OCL-Ausdrücke definiert, die die Korrektheit in Java-Projekten sicherstellen. So darf der Name von Paketen keinen Punkt enthalten, da der Punkt in `fullName`-Attributen dazu genutzt wird, die Beziehung zwischen Ober- und Unterpaketen darzustellen. Weiterhin wird gefordert, dass die vollständigen Namen von Paketen eindeutig sind und dass genau eine Wurzel pro Pakethierarchie existiert.

Darüber hinaus ist zu erwähnen, dass die Übereinstimmung von Paketen und Modulen der Softwarearchitektur (siehe Abschnitt 5.2.2) optional ist. Paketstrukturen werden zwar oft dazu genutzt, um die Softwarearchitektur im Programmcode abzubilden, dies wird jedoch ähnlich wie bei der `subPackage`-Beziehung nicht von Java gefordert oder erzwungen. Weiterhin kann durch Architekturdegradation (siehe Abschnitt 3.5) die Soll-Architektur von der im Quellcode tatsächlich vorhandenen Ist-Architektur abweichen. Seit der Einführung von Java 9 existiert ein eigenes Sprachkonstrukt, mit dem neben Paketen auch Module definiert werden können [MB17].

Typen Java-Typen werden innerhalb von Java-Artefakten definiert und durch Pakete strukturiert. Bei der Modellierung von Typen im AM wird nicht zwischen verschiedenen Arten von Typen unterschieden, d.h., durch die `Type`-Klasse des AMs lassen sich sowohl Java-Typen als auch Typen anderer (Modellierungs-) Sprachen repräsentieren. Von der weiteren Spezialisierung von Typen wurde abgesehen, da diese Details für artefaktbasierten Analysen nicht benötigt werden. Listing 6.9 zeigt den zugehörigen Ausschnitt des AMs.

Java-Typen lassen sich selbst weiter in Klassen, Interfaces oder Enums spezialisieren. Da aber bereits keine Unterscheidung zwischen Java-Typen und anderen Typen getroffen wurde, wird auch hier von einer weiteren Spezialisierung abgesehen. Somit wird bei der Modellierung des AMs ein einheitliches Abstraktionslevel verwendet, bei dem zwar teilweise auch Artefaktinhalte betrachtet werden, dort aber von Details weitestgehend abstrahiert wird.

Die `Type`-Klasse besitzt das `simpleName`-Attribut zur Beschreibung des Kurznamens sowie drei abgeleitete Attribute: `name`, `fullname` und `isInnerType`. Der Name wird aus dem Namen des beinhaltenden Typen und dem Kurznamen gebildet, während das `fullName`-Attribut den vollständigen Namen unter Einbeziehung des Paketnamens repräsentiert.

Jedes Java-Artefakt definiert eine beliebige Anzahl an Java-Typen. Jeder Typ kann außerdem durch mehrere Artefakte definiert werden (z.B. durch eine Java-Quellcodedatei und eine .class-Datei). Dies wird durch die `defines`-Beziehung (Zeile 8) modelliert.

Weiterhin ist ein Typ in genau einem Paket enthalten, während Pakete eine beliebige Anzahl an Typen enthalten können [GJS+15]. Dies wird durch die `contains`-Komposition in Zeile 10 definiert.

Typen können innere Typen enthalten, was durch die reflexive Assoziation der `Type`-Klasse modelliert wird. Die Anzahl an inneren Typen ist dabei beliebig, wobei jeder Typ in maximal einem anderen Typen enthalten sein kann. Diese Beziehung spiegelt sich bei der Namensbildung

CD
«AM»

```
1  class Type {
2    String simpleName;
3    /String name;
4    /String fullName;
5    /boolean isInnerType;
6  }
7
8  association defines [*] JavaArtifact -> Type [*];
9
10 composition contains [1] Package -> Type [*];
11
12 composition
13   [0..1] Type (containingType) -> (innerType) Type [*];
14
15 association reliesOn [*] Type -> Type [*];
16
17 context Type inv:
18   (isInnerType <=> !containingType.isAbsent) &&
19   (isInnerType implies
20     {t in containingType** | t.isInnerType}.size == 1);
21
22 context Type inv:
23   name == (isInnerType ?
24     containingType.name + "." + simpleName : simpleName);
25
26 context Type inv:
27   fullName == (package.isRoot ? name :
28     package.fullName + "." + simpleName);
```

Listing 6.9: AM: `Type`.

von inneren Typen wieder (in den Attributen `name` bzw. `fullname`), wodurch sich innere Typen per Name referenzieren lassen.

Typen können außerdem auf andere Typen angewiesen sein, was durch die `reliesOn`-Beziehung (Zeile 15) modelliert wurde. Ein Typ `A` ist auf einen Typ `B` angewiesen gdw. `B` für die Definition von `A` genutzt wird. Eine solche Nutzung kann auch bei der Definition von Elementen von `A` (Methoden, Konstanten, Variablen, Attributen, etc.) stattfinden. In der `reliesOn`-Beziehung werden alle verschiedenen Arten von Beziehungen aus [PKB13, PvdW15] zusammengefasst.

Der erste OCL-Ausdruck aus Listing 6.9 gibt an, dass innere Typen einen beinhaltenden Typ (`containingType`) haben. Die weiteren Ausdrücke definieren, wie der Name und der vollständige Name berechnet werden.

In Listing 6.10 werden einige zusätzliche OCL-Ausdrücke modelliert, welche die Zusammenhänge zwischen Java-Artefakten und Java-Typen unter Betrachtung von Ordnern und Paketen genauer spezifizieren.

CD
«AM»

```
1  context JavaSourceFile inv:
2    defines == compiledTo.defines;
3
4  context JavaSourceFile inv:
5    {t in defines | !t.isInnerType}.size == 1;
6
7  context JavaArtifact a1, a2,
8    Type t1 in a1.defines, Type t2 in a2.defines inv:
9      t1.reliesOn.contains(t2) implies
10       (a1 == a2 || a1.reliesOn.contains(a2));
11
12 context JavaArtifact a, Type t in a.defines inv:
13   !t.isInnerType implies t.simpleName == a.simpleName;
14
15 context JavaArtifact inv:
16   parent.fullName.replaceAll("/", ".")
17     .endsWith(belongsTo.fullName);
18
19 context JavaSourceFile inv:
20   forall n in {t.fullName.replaceAll(".", "/") |
21     t in defines, !t.isInnerType}:
22       fullName.endsWith(n + ".java");
23
24 context JavaClassFile inv:
25   parent.fullName.replaceAll("/", ".")
26     .endsWith(defines.package.fullName);
27
28 context JavaClassFile inv:
29   {simpleName} == defines.name.replaceAll(".", "$");
```

Listing 6.10: AM: Beziehungen zwischen Java-Artefakten und -Typen.

Zeile 1 beschreibt den Zusammenhang zwischen den Typen von Java-Quellcodedateien und .class-Dateien. Die in Java-Quellcodedateien definierten Typen sind mit den definierten Typen aller .class-Dateien, die aus der Quelldatei beim Kompiliervorgang hervorgegangen sind, identisch.

Zeile 4 stellt sicher, dass jede Java-Quellcodedatei nur genau einen Top-Level-Typ definiert. Diese Richtlinie wurde zur Sicherstellung einer systematischen und einfachen Programmierung modelliert und wird nicht vom Java-Compiler gefordert. Dieser kann auch

Java-Quellcodedateien verarbeiten, in denen mehrere oder keine Top-Level-Typen definiert wurden.

Zeile 7 besagt, dass, wenn ein Java-Typ `A` auf einen Java-Typ `B` angewiesen ist, beide Typen entweder im gleichen Artefakt definiert wurden oder die beiden Artefakte analog zu ihren Typen aufeinander angewiesen sind.

Die weiteren OCL-Ausdrücke beschränken die Namen von Java-Artefakten und deren Typen. Durch die Prüfbedingung in Zeile 12 wird festgelegt, dass der Kurzname eines Java-Artefakts und der Kurzname des durch das Artefakt definierten Top-Level-Typs identisch sind. Des Weiteren wird durch die Prüfbedingung aus Zeile 15 festgelegt, dass der Name eines Ordners, der eine Java-Quellcodedatei enthält, zu dem Paketnamen dieser Java-Datei passen muss. Diese Bedingung wird nicht durch den Java-Compiler erzwungen, dafür aber von anderen Entwicklungswerkzeugen, wie beispielsweise der Eclipse IDE [Ecl17]. Bei der Spezifikation der Bedingung mussten die verschiedenen Separatoren (’.’ bei Typen und ’/’ bei Dateien) entsprechend angeglichen werden. Eine entsprechende Anpassung wird auch bei den Spezifikationen der restlichen OCL-Ausdrücke angewendet. Durch Zeile 19 wird sichergestellt, dass der vollständige Name von Java-Quellcodedateien zu dem vollständigen Namen des durch die Datei definierten Top-Level-Typen passt. In Zeile 24 wird definiert, dass eine kompilierte .class-Datei in einem Ordner liegt, dessen vollständiger Name den Paketnamen der zugehörigen Quellcodedatei reflektiert. Die letzte Prüfbedingung aus Listing 6.10 ist in Zeile 28 dargestellt und gibt an, dass der Kurzname von .class-Dateien dem Namen des durch die Datei definierten Typen entspricht.

6.2. Definition von Modellierungssprachen

Modellierungssprachen, wie beispielsweise die Sprachen der UML/P (siehe Abschnitt 2.4.1) benötigen eine geeignete Definition, damit sie für Menschen gut benutzbar sind, aber auch maschinell verarbeitet werden können. Eine gute Möglichkeit sind an dieser Stelle textuelle Modellierungssprachen, die sich unter anderem mit MontiCore (vgl. Abschnitt 2.3) definieren lassen. Grammatiken [KRV07a, KRV07b] werden als Teil der Sprachdefinition verwendet. Diese Art der Definition wird im Folgenden angenommen. Alternativ dazu können auch Metamodelle wie MOF [OMG16] oder Ecore [SBMP08] zur Definition von Sprachen verwendet werden. Die Sprachdefinitionen werden in verschiedenen Artefakten gespeichert, die wiederum durch das AM modelliert werden können.

Der in diesem Abschnitt beschriebene Teil des AMs beschreibt Artefakte sowie verwandte Konzepte, die bei der Definition von Modellierungssprachen eingesetzt werden. Somit dient dieser Teil als Grundlage, um später konkrete Modellierungssprachen (siehe Abschnitt 6.3) und Generatoren (siehe Abschnitt 6.4) beschreiben zu können. Abbildung 6.11 gibt einen Überblick über die Konzepte und Artefakte, die zur Definition von Modellierungssprachen verwendet werden. Diese werden in den folgenden Abschnitten detaillierter spezifiziert.

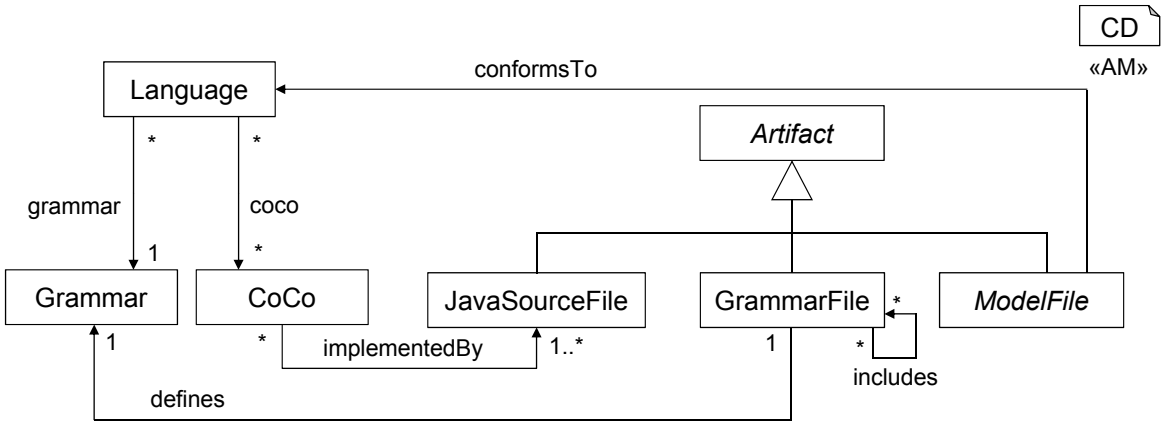

Abbildung 6.11.: Eine Sprache wird durch eine Grammatik und mehrere Wohlgeformtheitsregeln beschrieben.

6.2.1. Modellierungssprachen

Die wichtigsten Konzepte bei der Definition von Modellierungssprachen werden durch Listing 6.12 definiert. Diese sind die Sprache selbst (`Language`), die zugehörige Grammatik (`Grammar`) sowie CoCos (`CoCo`). Sie stellen keine Artefakte dar, sondern die logischen Inhalte von Artefakten wie bereits in Abbildung 6.11 dargestellt wurde.

CD «AM»

```
1  class Language {
2    String name;
3  }
4
5  class Grammar {}
6
7  class CoCo {
8    String name;
9  }
10
11 association grammar [*] Language -> Grammar [1];
12
13 association coco [*] Language -> CoCo [*];
14
15 context Language inv:
16   name != "";
```

Listing 6.12: AM: `Language`.

Die `Language`-Klasse des AMs repräsentiert jede Art von Modellierungs- und Programmiersprache sowie andere formale Sprachen. Da hier von rein textuellen Sprachen ausgegangenen wird, lassen sich diese durch Grammatiken ausdrücken. Zur Definition einer Sprache

wird dabei genau eine Grammatik verwendet. Um weitere Details zur Definition der Wohlgeformtheit zugehöriger Modelle zu spezifizieren, können zusätzlich `CoCos` verwendet werden. Eine CoCo beschränkt die Sprache durch weitere Bedingungen, die von zugehörigen Modellen eingehalten werden müssen.

6.2.2. Grammatiken

In MontiCore und anderen Language Workbenches, wie beispielsweise Xtext [EB10], Spoofax [KV10] oder MPS [Voe13], ist es möglich Grammatiken in Artefakten abzuspeichern. Im AM wird dies durch die Klassen `Grammar` als logisches Konzept sowie `GrammarFile` als zugehöriges Artefakt reflektiert. Grammatikdateien definieren dabei genau eine Grammatik, können zu deren Definition aber auf andere Grammatikdateien zurückgreifen. Dies wird durch die `includes`-Beziehung dargestellt. In MontiCore ist es beispielsweise möglich, Nichtterminale aus inkludierten Grammatiken zu erweitern und zu ersetzen sowie verschiedene Grammatiken zu kombinieren und zu verfeinern.

CD
«AM»

```
1  class GrammarFile extends Artifact {}
2
3  association defines [1] GrammarFile -> Grammar [1];
4
5  association includes [*] GrammarFile -> GrammarFile [*];
6
7  context GrammarFile inv:
8    nameExtension == "mc4";
9
10 context GrammarFile inv:
11   refersTo.containsAll(includes);
```

Listing 6.13: AM: `GrammarFile`.

In Listing 6.13 wird die Klasse `GrammarFile` sowie die zugehörigen Beziehungen modelliert. Darüber hinaus wird durch OCL-Ausdrücke zum einen die Dateiendung von Grammatikdateien festgelegt („mc4“) und zum anderen angegeben, dass die `includes`-Beziehung zwischen Grammatikdateien eine Spezialisierung der `refersTo`-Beziehung darstellt.

CoCos werden mit Hilfe von Java-Quellcodedateien implementiert (`implementedBy`). Dabei ist sowohl die Definition mehrerer CoCos in einer einzelnen Java-Datei als auch die Definition einer CoCo in mehreren Java-Dateien oder eine Mischform möglich. Der entsprechende Ausschnitt des AMs ist in Listing 6.14 dargestellt.

```
1  association implementedBy [*] CoCo -> JavaSourceFile [1..*];
```
CD «AM»

Listing 6.14: AM: `CoCo`.

6.2.3. Modelldateien

Durch den Einsatz von Modellierungssprachen wird *Entwicklern* die Möglichkeit zur Modellierung gegeben. Modelle werden dabei als Artefakte gespeichert. In Listing 6.15 wird mit der Klasse `ModelFile` das entsprechende Element im AM eingeführt.

```
1  abstract class ModelFile extends Artifact {}
2
3  association conformsTo [*] ModelFile -> Language [*];
```
CD «AM»

Listing 6.15: AM: `ModelFile`.

Modelldateien können einer Sprache entsprechen. Dabei ist es natürlich möglich, dass für eine Sprache mehrere solcher Modelldateien existieren. Umgekehrt ist es aber ebenfalls möglich, dass eine Modelldatei den Regeln mehrerer Sprachen entspricht. Dies ist vor allem dann der Fall, wenn eine Sprache in mehreren Sprachversionen existiert. Ein Beispiel aus der Programmierung sind Java-Dateien, die die Regeln mehrerer Java-Versionen (z.B. Java 7 und 8) erfüllen. Demgegenüber existieren jedoch auch Artefakte, die die zugehörige Sprache inklusive der Versionsnummer im Artefakt abspeichern, wie es beispielsweise bei XML-Dateien vorkommt. Diese Artefakte gehören dann nur zu genau einer Version der Sprache.

Bei der Spezifikation von Modelldateien im AM wurde auf eine noch detailliertere Modellierung verzichtet. Auch an dieser Stelle hätte zwischen Modell als Konzept und dem zugehörigen Artefakt unterschieden werden können. Da jedoch nur die Grundlage für die konkrete Modellierung in späteren Abschnitten bereitgestellt werden soll und dieser Abstraktionsgrad ausreicht, ist die Entscheidung auf die dargestellte Modellierung gefallen. Eine Erweiterung, die für zukünftige Arbeiten interessant sein könnte, ist die Modellierung der Zusammenhänge von Modellen, Modellelementen und deren Verarbeitung durch MontiCore-basierte MDD-Werkzeuge (siehe auch Abschnitt 11.2). Die Betrachtung der Überführung einzelner Modellelemente in die speicherinterne AST-Repräsentation würde ein feingranulareres Tracing der Abläufe solcher Werkzeuge ermöglichen.

6.3. Klassendiagramm Artefakte

In Abschnitt 6.1 wurde bereits der Ausschnitt aus dem AM für Java-Artefakte vorgestellt. Das AM soll darüber hinaus auch die Analyse von MDD-Projekten ermöglichen. Somit reicht es nicht aus, nur die Artefakte einer typischen Programmiersprache beschreiben zu können,

sondern es müssen auch Artefakte und Konzepte von spezifischen Modellierungssprachen unterstützt und somit bei der artefaktbasierten Analyse mitbetrachtet werden können. Durch die Modellierung aus Abschnitt 6.2 wurden bereits Vorbereitungen für die Abbildung konkreter Modellierungssprachen getroffen.

In diesem Abschnitt folgt nun die Modellierung von CD-Artefakten und zugehörigen Konzepten. Zukünftige Erweiterungen des AMs und Neuerstellungen projektspezifischer AMe (siehe Abschnitt 7.1) können sich an der in diesem Abschnitt vorgestellten Modellierung orientieren, um die Analyse von Artefakten anderer Modellierungssprachen zu unterstützen.

UML/P-CDs [Rum16, Rum17] sind Strukturdiagramme, die zu verschiedenen Zwecken eingesetzt werden können (siehe Abschnitt 2.4.1). In vielen MDD-Projekten werden CDs dazu verwendet, die Konzepte der Problemdomäne durch konzeptionelle Modelle abzubilden. Diese Modelle können durch den Einsatz geeigneter Werkzeuge für eine direkte Weiterverarbeitung in den MDD-Buildprozess eingebunden werden. Dabei dienen sie als Eingabe für Generatoren [Rei16, Rot17, Loo17], die aus den modellierten Informationen Datenmanagementsysteme inklusive GUI und Persistenzfunktionalität generieren. Die Beziehung zwischen Eingabemodellen und generierten Artefakten wird später in Abschnitt 6.4 näher untersucht. Abbildung 6.16 gibt eine Übersicht über die Klasse `CDModelFile` sowie deren verwandte Elemente und Beziehungen.

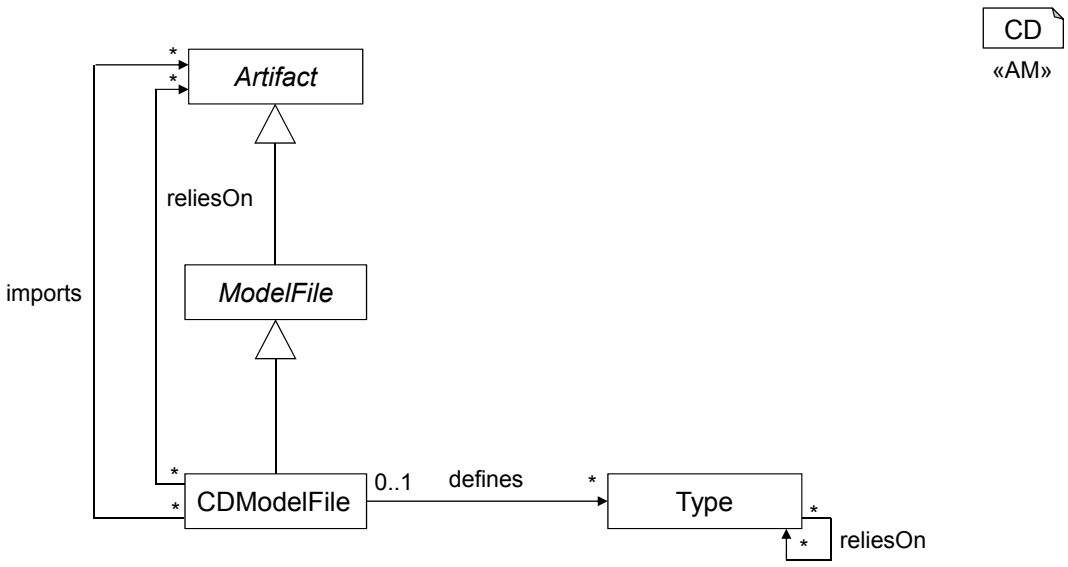

Abbildung 6.16.: CD-Modelldateien, verwandte Elemente und Beziehungen.

Ein `CDModelFile` ist ein Artefakt, das ein CD in textueller Notation [Sch12] definiert. CD-Modelldateien importieren andere Artefakte entsprechend ihrer Importstatements, wobei Importe dazu führen, dass Symbole der importierten Artefakte zur Definition des CDs genutzt werden können. MontiCore stellt mit seinen Möglichkeiten zur Sprachkomposition [HLMSN+15a] und der Nutzung der Symboltabelle [MSN17] zum Auflösen von Importen eine

geeignete Infrastruktur bereit. Im Falle von CD-Modelldateien können importierte Symbole auch durch Artefakte anderer Sprachen bereitgestellt werden.

An dieser Stelle des AMs könnte ebenfalls, wie auch bei der Modellierung von Java-Artefakten und den verwandten Konzepten, eine wesentlich detailliertere Modellierung der syntaktischen Struktur der Artefakte stattfinden, wodurch zwischen verschiedenen Typarten (Klassen, Interfaces, Enums) und Beziehungen (Assoziationen, Aggregationen und Kompositionen) unterschieden werden könnte. Da auch hier das Ziel ist, von den Artefaktinterna zu abstrahieren, werden Details ausgelassen, sofern sie nicht für die artefaktbasierte Analyse notwendig sind.

Dennoch werden in diesem Kapitel neben CD-Modellartefakten auch Typen betrachtet. Diese wurden bereits durch Listing 6.9 definiert. Die Spezifikation von CD-Modelldateien wird in Listing 6.17 dargestellt.

CD
«AM»

```
1  class CDModelFile extends ModelFile {}
2
3  association imports [*] CDModelFile -> Artifact [*];
4
5  association reliesOn [*] CDModelFile -> Artifact [*];
6
7  association defines [0..1] CDModelFile -> Type [*];
8
9  context CDModelFile inv:
10   nameExtension == "cd";
11
12 context CDModelFile inv:
13   reliesOn.containsAll(imports);
14
15 context CDModelFile inv:
16   refersTo.containsAll(reliesOn);
17
18 context CDModelFile m1, m2,
19   Type t1 in m1.defines, Type t2 in m2.defines inv:
20     t1.reliesOn.contains(t2) implies
21       (m1 == m2 || m1.reliesOn.contains(m2));
22
23 context CDModelFile m, JavaArtifact a,
24   Type t1 in m.defines, Type t2 in a.defines inv:
25     t1.reliesOn.contains(t2) implies m.reliesOn.contains(a);
```

Listing 6.17: AM: `CDModelFile`.

Die `imports`-Beziehung wird dort als Spezialisierung der `reliesOn`-Beziehung modelliert, während die `reliesOn`-Beziehung ihrerseits als Spezialisierung der `refersTo`-Beziehung definiert wird. Darüber hinaus wird durch die Prüfbedingungen in den Zeilen 18

und 23 sichergestellt, dass alle benötigten Typen, auf die die zugehörige CD-Modelldatei angewiesen ist, durch Artefakte bereitgestellt werden.

Das AM ist so gestaltet, dass Java-Artefakte im Gegensatz zu CD-Modelldateien nur auf andere Java-Artefakte angewiesen sind und nicht auf Typen aus anderen Artefakten zugreifen können (vgl. Listing 6.9). Eine entsprechende Erweiterung könnte dann notwendig werden, wenn dargestellt werden soll, dass handgeschriebene Java-Klassen mit generierten Java-Klassen kooperieren, wobei die generierten Java-Klassen die Implementierung von Typen eines CDs darstellen. Auf Modellebene kann dann von den generierten Dateien abstrahiert werden und stattdessen die Beziehungen zwischen den handgeschriebenen Java-Quellcodedateien und CD-Modelldateien betrachtet werden. Die beschriebene Situation entsteht bei der Verwendung von CD-zu-Java-Generatoren, wie beispielsweise denen aus [Rei16, Loo17, Rot17]. Das AM kann als Basis für ähnliche Erweiterungen genutzt werden, um somit auch komplexere Beziehungsgeflechte von Artefakten anderer Modellierungssprachen abbilden zu können.

6.4. Generatorartefakte

Generatoren lesen mehrere existierende Artefakte ein und produzieren während des Ablaufs eine beliebige Anzahl an Ausgabeartefakten. Dabei werden als Eingabe von Generatoren oftmals Modellartefakte und als Ausgabe Quellcodeartefakte angenommen, die nach dem Kompilieren ein lauffähiges System darstellen. Somit führt ein Generator eine M2T-Transformation durch, die Modelle in ein lauffähiges Programm transformiert. Die eingelesenen Eingabemodelle gehören dabei jeweils zu einer wohldefinierten Eingabesprache. Im AM wird als Ausgabesprache für generierte Quellcodedateien stets Java angenommen. Zwar liegt der Fokus bei der Modellierung von Generatoren auf MontiCore-basierten Technologien, was aber nicht zwingend eine Einschränkung darstellt, da auch einige andere Generatortechnologien ähnliche Ansätze verwenden. MontiCore-basierte Generatoren nutzen templatebasierte Codegenerierung (siehe Abschnitt 2.3.3). Dabei wird die Templateengine FreeMarker [Fre17] verwendet. FreeMarker erlaubt ein gegenseitiges Aufrufen von Templates. Darüber hinaus können Templates während der Generierung Java-Objekte erstellen und deren Methoden aufrufen. Somit besteht ein Generator unter anderem aus FreeMarker-Templates und Java-Artefakten. Eine Übersicht über den zugehörigen AM-Teil wird in Abbildung 6.18 gegeben.

Mit Abbildung 6.18 werden hauptsächlich die Beziehungen zwischen Generatoren, FreeMarker-Templates und Java-Artefakten eingeführt. Die Beziehungen zwischen FreeMarker-Templates und Java-Artefakten treten deshalb auf, weil Java nicht nur als Sprache für den generierten Quellcode sondern gleichzeitig als Programmiersprache zur Implementierung von Generatoren genutzt wird. Weitere Details von FreeMarker-Templates werden in Abschnitt 6.4.1 vorgestellt.

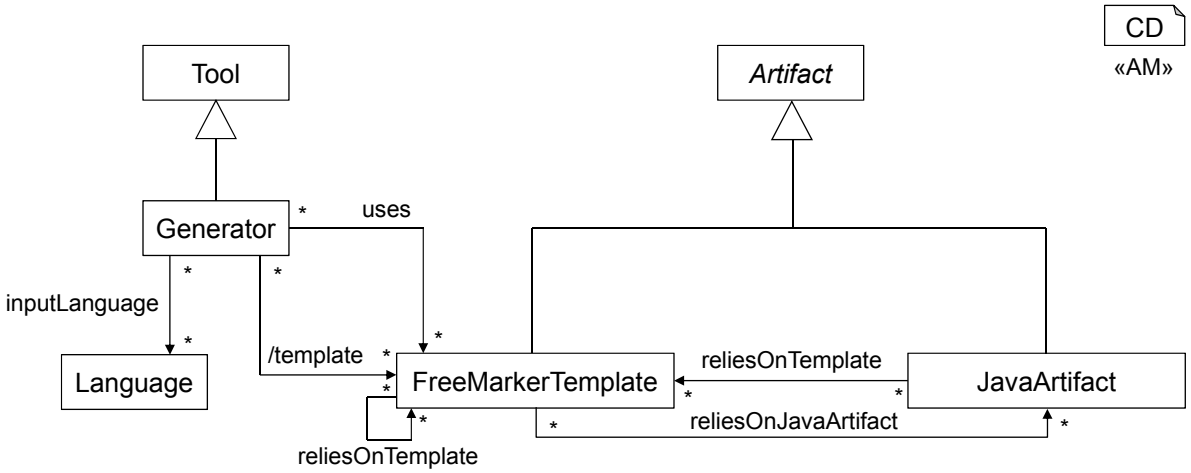

Abbildung 6.18.: Generatoren sind Werkzeuge, die zu den Eingabesprachen passende Modelle einlesen und unter anderem aus FreeMarker-Templates und Java-Artefakten bestehen.

6.4.1. Templates

Bevor Generatoren als Ganzes im Detail beschrieben werden können, wird in diesem Abschnitt die Modellierung von Templates als wichtige Generatorartefakte beschrieben. Durch den Einsatz von Templates wird eine komfortable Generierung von Quellcode möglich. Dazu werden Templates zu großen Anteilen im Zielformat zuzüglich variabler Anteile, die zur Laufzeit basierend auf den Eingaben des Generators befüllt werden, spezifiziert (vgl. Abschnitt 2.3.3). In Listing 6.19 wird zunächst die `Template`-Klasse zusammen mit ihren statisch rekonstruierbaren Beziehungen angegeben. Die Beziehungen, die sich zur Laufzeit von Generatoren beobachten lassen folgen dann in Abschnitt 6.5.

Wie oben erwähnt, können sich Templates bei ihrer Ausführung gegenseitig aufrufen [Sch12], was dazu führt, dass Templates auf andere Templates angewiesen sind. Diese Beziehung wird durch die `reliesOnTemplate`-Beziehung des AMs modelliert.

Weiterhin ist es möglich Java-Objekte aus Templates heraus zu erzeugen, sie zwischenzuspeichern, als Parameter an andere Templates zu übergeben und deren Methoden aufzurufen. In den Fällen, wo eine Instanziierung oder ein normaler Methodenaufruf stattfindet, sind Templates auf das Vorhandensein der entsprechenden Java-Artefakte angewiesen. Diese Beziehung wird im AM durch die `reliesOnJavaArtifact`-Beziehung dargestellt.

Mit der `reliesOnTemplate`-Beziehung von Java-Artefakten zu FreeMarker-Templates existiert ein Gegenstück zu der `reliesOnJavaArtifact`-Beziehung. Java-Artefakte können auf die Anwesenheit von Templates angewiesen sein, wenn sie standardmäßig vorgesehene Templates durch projektspezifische Templates ersetzen [Rot17]. In den zugehörigen Java-Statements werden Templates per qualifiziertem Namen referenziert.

Damit ist in MontiCore-basierten Werkzeugen ein flexibles Management von Templates möglich, da sie sich einerseits gegenseitig aufrufen können, wodurch eine Templatehierarchie

CD
«AM»

```
class FreeMarkerTemplate extends Artifact {}

association reliesOnTemplate
  [*] FreeMarkerTemplate -> FreeMarkerTemplate [*];

association reliesOnJavaArtifact
  [*] FreeMarkerTemplate -> JavaArtifact [*];

association reliesOnTemplate
  [*] JavaArtifact -> FreeMarkerTemplate [*];

context FreeMarkerTemplate inv:
  nameExtension == "ftl";

context FreeMarkerTemplate inv:
  refersTo.containsAll(reliesOnTemplate);

context FreeMarkerTemplate inv:
  refersTo.containsAll(reliesOnJavaArtifact);

context JavaArtifact inv:
  refersTo.containsAll(reliesOnTemplate);
```

Listing 6.19: AM: `FreeMarkerTemplate` und dessen Assoziationen.

entsteht und andererseits durch die Möglichkeit des Ersetzens von Templates aus Java heraus ein mächtiger Konfigurationsmechanismus existiert. In [GMR+16] wurde ein weiterführender Ansatz für die Konfiguration von Templates innerhalb von Generatoren vorgestellt, welcher die Verwendung einer speziellen DSL zu Konfigurationszwecken vorsieht.

Durch die statische Rekonstruktion der vorgestellten Beziehungen wird eine Untersuchung der Templatestruktur möglich. Jedoch ist diese Rekonstruktion nicht trivial, da Referenzen auf Templates durch Strings angegeben werden und somit erst zur Laufzeit berechnet werden können. Diese Art der Berechnung wird nicht gänzlich verboten, sondern als Teil des Parametrisierungs- und Konfigurationsmechanismus akzeptiert. Allerdings sollte von diesen Mechanismen nur spärlich Gebrauch gemacht werden, da in Kauf genommen werden muss, dass eine statische Extraktion der Beziehung nicht mehr möglich ist. Darüber hinaus wird das Verständnis von Generatoren erschwert, wenn Strings, die Referenzen auf Templates darstellen, zur Laufzeit manipuliert werden. Im Gegensatz zu der statischen Extraktion ist aber eine Beobachtung solcher Fälle zur Laufzeit weiterhin einfach möglich.

6.4.2. Generatoren

Generatoren sind Systeme und bestehen somit auch aus Artefakten. Zusätzlich zu der `contains`-Beziehung aus Listing 5.9 werden für Generatoren die verschiedenen Beziehungen zu Templates separat angegeben, da sie von besonderem Interesse für die artefaktbasierte Analyse sind. Listing 6.20 definiert die `Generator`-Klasse des AMs zusammen mit ihren Assoziationen und einigen OCL-Ausdrücken.

CD
«AM»

```
1  class Generator extends Tool {}
2
3  association uses [*] Generator -> FreeMarkerTemplate [*];
4
5  association /template
6    [*] Generator -> FreeMarkerTemplate [*];
7
8  association inputLanguage [*] Generator -> Language [*];
9
10 context Generator g, JavaArtifact a in g.consistsOf inv:
11   g.template == g.uses.addAll(g.template.reliesOnTemplate**)
12                       .addAll(a.reliesOnTemplate);
13
14 context Generator inv:
15   consistsOf.contains(template);
16
17 context Generator inv:
18   consistsOf.contains(template.reliesOnJavaArtifact);
```

Listing 6.20: AM: `Generator`.

Wie erwähnt nutzen Generatoren Templates zum Zweck der Codegenerierung. In Listing 6.20 wird dabei zwischen zwei verschiedenen Assoziationen unterschieden: `uses` und `template`. Durch die `uses`-Beziehung werden nur die vom Generator direkt aufgerufenen Templates repräsentiert. Diese Templates können dadurch identifiziert werden, dass sie beim Aufruf der Templateengine (siehe Abschnitt 2.3.1) als Parameter übergeben werden. Demgegenüber vereinigt die `template`-Beziehung diese Menge mit allen Templates, die dem Generator nur indirekt bekannt sind. Solche Templates werden entweder durch direkt genutzte Templates eingebunden oder es sind Templates die von Java-Artefakten des Generators benötigt werden. Somit stellt die `template`-Beziehung die vollständige, für die Generierung notwendige Menge an Templates bereit. Ob alle Templates bei einer konkreten Ausführung des Generators tatsächlich benötigt werden, kann nicht statisch bestimmt werden, da dies von der Eingabe des Generators abhängt. Welche Arten von Eingabemodellen zulässig sind, wird durch die spezifizierten Eingabesprachen (`inputLanguage`) festgelegt.

In Abbildung 5.2 wird bereits durch die `produces`-Beziehung angegeben, dass Artefakte zur Erstellung anderer Artefakte beitragen können. Bei Ausführung von Generatoren tragen beteiligte Template- und Java-Artefakte zur Erstellung generierter Ausgabeartefakte bei. Um diese Arten von Beziehungen genauer spezifizieren zu können, müssen die Abläufe bei der Generierung mitbetrachtet werden. Dazu kann die in Abschnitt 6.5 folgende Modellierung von Generierungsausführungen genutzt werden.

6.5. Dynamische Beziehungen zur Generierungszeit

Systemabläufe werden im AM mit Hilfe von Events und Aktionen beschrieben. Während ein Event nur einmalig und ohne Dauer auftritt, kann die Ausführung einer Aktion eine gewisse Zeit in Anspruch nehmen. In diesem Kapitel wird die Modellierung von dynamischen Abläufen im AM vorgestellt. Dazu werden zunächst die abstrakten Klassen `Action` und `Event` eingeführt, die im Verlauf des Abschnitts durch konkrete Klassen spezialisiert werden.

6.5.1. Aktionen und Events

In Abbildung 6.21 wird eine Übersicht über die Modellierung von Aktionen und Events sowie deren Beobachtung gegeben. Eine detaillierte Spezifikation folgt anschließend in Listing 6.22.

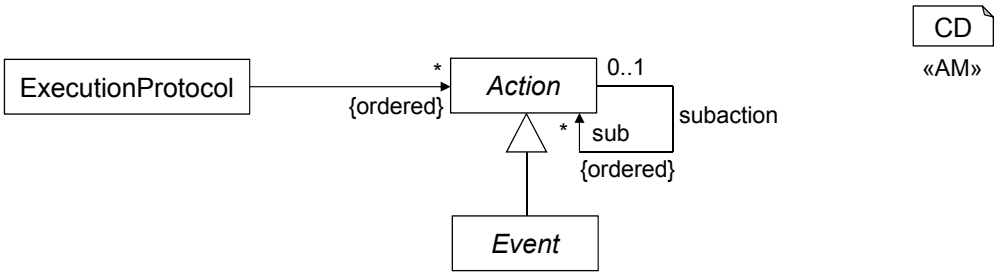

Abbildung 6.21.: Aktionen, Events und deren Beobachtung.

Das `ExecutionProtocol` ist ein logisches Konzept, das alle Aktionen eines Durchlaufs in geordneter Reihenfolge speichert. Ein solches Protokoll kann direkt in eine Datei gespeichert werden. Aktionen werden durch die Klasse `Action` repräsentiert, die mit `start` und `end` zwei Attribute enthält, um die Zeitstempel für die Erfassung der Dauer der Aktion festzuhalten.

Durch die OCL-Prüfbedingungen aus Listing 6.22 werden die möglichen Werte der beiden Attribute eingeschränkt. In Zeile 11 wird sichergestellt, dass der Startzeitpunkt einer Aktion nicht nach dem Endzeitpunkt liegt und durch Zeile 14 wird ausgedrückt, dass Events keine Dauer haben und somit deren Start- und Endzeitpunkte identisch sein müssen.

Eine Aktion kann selbst aus Unteraktionen bestehen, die auch Events sein können. Die zugehörige Assoziation (`subaction`) ist geordnet, sodass die Reihenfolge, in der Unteraktionen auftreten, festgehalten wird. Auch für diese Beziehung wurde eine weitere Einschränkung der Start- und Endzeitpunkte von Aktionen modelliert. Dazu wird durch Zeile 17 festgelegt, dass

CD «AM»

```
class Action {
  Date start;
  Date end;
}

class Event extends Action {}

association subaction
  [0..1] Action (parent) -> (sub) Action [*] <<ordered>>;

context Action inv:
  start <= end;

context Event inv:
  start == end;

context Action a, Action s in a.subaction inv:
  a.start <= s.start && s.end <= a.end;
```

Listing 6.22: AM: `Actions`.

der Startzeitpunkt jeder Unteraktion nicht vor dem Startzeitpunkt der Oberaktion und außerdem der Endzeitpunkt jeder Unteraktion nicht nach dem Endzeitpunkt der Oberaktion liegen darf.

Listing 6.23 definiert die Klasse `ExecutionProtocol` mit deren Hilfe ein sequentieller Prozess modelliert wird.

CD «AM»

```
class ExecutionProtocol {}

association ExecutionProtocol -> Action [*] <<ordered>>;

context ExecutionProtocol p inv:
  p.action.nonEmpty implies (p.action.asSet ==
    p.action[0].subaction**.add(action[0]));

context ExecutionProtocol p, int i,k inv:
  (0 <= k && k <= p.action.size && 0 <= i && i <= k) implies
    p.action[i].start <= p.action[k].start;

context ExecutionProtocol p, int i,k inv:
  (0 <= k && k <= p.action.size && 0 <= i && i <= k) implies
    p.action[i].end <= p.action[k].start;
```

Listing 6.23: AM: `ExecutionProtocol`.

Parallele Abläufe im Buildprozess können durch diese Modellierung nicht abgebildet werden. Das `ExecutionProtocol` ist mit der flachen Liste aller, bei Prozessdurchführung beobachteter Aktionen (inklusiv der Unteraktionen), verknüpft. Um sicherzustellen, dass auch alle Unteraktionen in der flache Liste des Protokolls enthalten sind, wurde die Bedingung in Zeile 5 modelliert. In Zeile 9 und Zeile 13 wird weiterhin sichergestellt, dass die Reihenfolge der Aktionen durch ihre Zeitstempel festgelegt ist.

6.5.2. Aktionen des Generierungsprozesses

Nachdem in Abschnitt 6.5.1 bereits die abstrakte Modellierung von Aktionen im AM vorgestellt wurde, werden in diesem Abschnitt konkrete, für den Generierungsprozess relevante Aktionen eingeführt. In Abbildung 6.24 wird eine Übersicht über die bei Ablauf des Generierungsprozesses relevanten Aktionen gegeben. Eine detaillierte Spezifikation dieses AM-Ausschnitts findet sich in Listing 6.25.

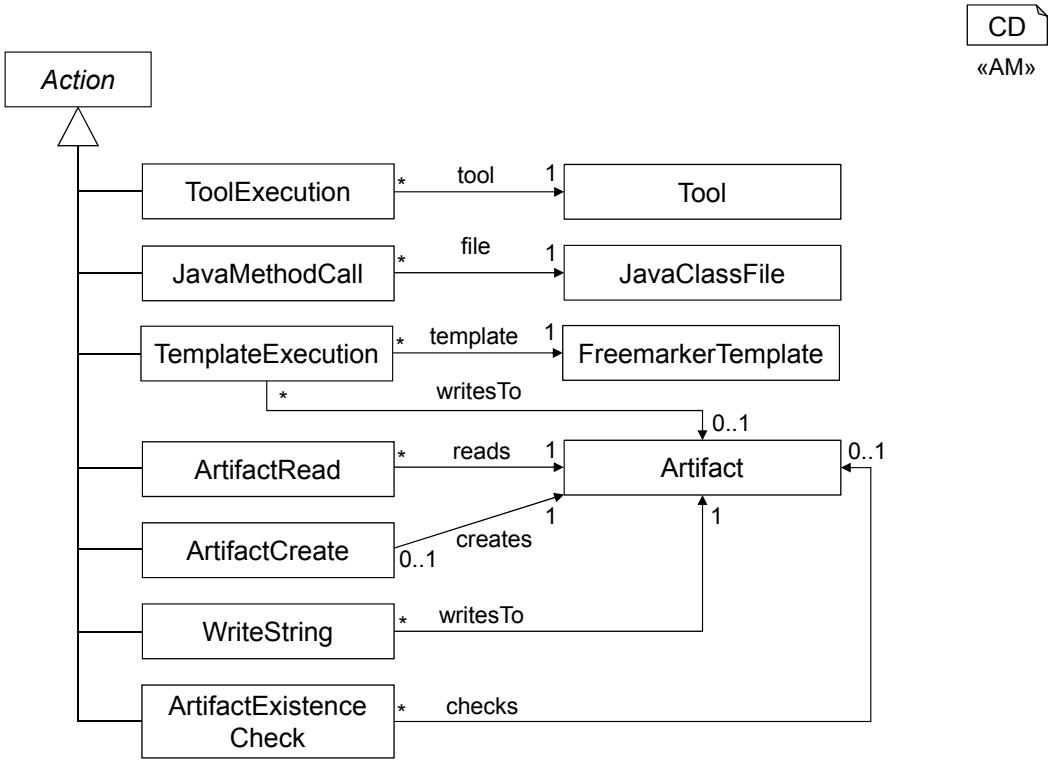

Abbildung 6.24.: Relevante Aktionen des Generierungsprozesses.

Bei der Generierung spielen Templates eine wichtige Rolle. Ein Verständnis der genutzten Templates und deren Beitrag bei der Generierung führt zu einem besseren Verständnis des gesamten Generierungsprozesses. Aus diesem Grund sind templatebezogene Aktionen besonders interessant. Bei der Ausführung eines Templates (`TemplateExecution`) wird ein Template zunächst ausgewertet und der berechnete Text in ein Zielartefakt geschrieben. In diesen Fällen wird der Templateausführung das entsprechende Artefakt zugeordnet (`writesTo`). Es gibt

CD
«AM»

```
class ToolExecution           extends Action {}
class TemplateExecution       extends Action {}
class JavaMethodCall          extends Action {}
class ArtifactRead            extends Action {}
class ArtifactExistenceCheck  extends Action {
  String fullName;
}
class ArtifactCreate          extends Action {}
class WriteString             extends Action {
  String content;
}

association tool [*] ToolExecution -> Tool [1];

association writesTo
  [*] TemplateExecution -> Artifact [0..1];

association template
  [*] TemplateExecution -> FreeMarkerTemplate [1];

association file [*] JavaMethodCall -> JavaClassFile [1];

association reads [*] ArtifactRead -> Artifact [1];

association checks
  [*] ArtifactExistenceCheck -> Artifact [0..1];

association creates [0..1] ArtifactCreate -> Artifact [1];

association writesTo [*] WriteString -> Artifact [1];
```

Listing 6.25: AM: `Actions`.

aber auch Fälle, in denen Templates ausschließlich dazu genutzt werden, um die Abläufe bei der Generierung zu strukturieren. Solche Templates schreiben nicht selbst in eine Datei. Stattdessen sorgen sie dafür, dass weitere Unteraktionen ausgeführt werden.

Weiterhin sind die bei der Generierung gelesenen sowie die erstellten Artefakte von Interesse. MontiCore-basierte Generatoren können darüber hinaus auf die bloße Existenz von handgeschriebenen Artefakten reagieren. Durch die Aktionen `ArtifactRead`, `ArtifactCreate` und `ArtifactExistenceCheck` werden die zugehörigen Aktionen im AM beschrieben.

Mit `JavaMethodCall` wurde eine Aktion modelliert, die Java Methodenaufrufe repräsentiert. Bei der Beobachtung von Generierungsprozessen sind allerdings nicht zwingend alle Methodenaufrufe von Interesse. Aus diesem Grund sollten bei der Extraktion von Artefaktdaten

nur die für die jeweiligen artefaktbasierten Analysen relevanten Methodenaufrufe extrahiert werden. Ein Beispiel für besonders interessante Methodenaufrufe sind Aufrufe aus Templates heraus. Dies schließt auch die Instanziierung neuer Java-Objekte mit ein.

Die weiteren Aktionen aus Abbildung 6.24 sind `ToolExecution` und `WriteString`. Ersteres repräsentiert die einzelne Ausführung eines Werkzeugs, während durch Letzteres das Schreiben eines Strings in eine Datei modelliert wurde.

An dieser Stelle sei gesagt, dass es durchaus noch Erweiterungspotenzial bei der Beschreibung von MontiCore-basierten Buildprozessen gibt. Dies umfasst die Betrachtung der Ausführung von M2M-Transformationen [Wei12, HRW15, AHRW17, Höl18], die Erstellung des ASTs, die Erstellung und Nutzung der Symboltabelle [MSNRR16, MSN17] sowie die Ausführung von Visitoren [HMSNRW16].

Im Folgenden werden, basierend auf den hier vorgestellten, feingranularen Aktionen, zusätzlich die Beziehungen von Werkzeugen und Artefakten definiert, die ein Verständnis der Abläufe auf einem höheren Abstraktionsniveau ermöglichen.

6.5.3. Ein- und Ausgaben von Werkzeugen

Eines der Ziele bei der Beobachtung von Ein- und Ausgabeartefakten bei Ausführung von Generatoren oder ganzer MDD-Werkzeugketten ist, bei Änderungen an den Eingaben den Teil des Buildprozesses zu identifizieren, der erneut ausgeführt werden muss. Dadurch werden effiziente und inkrementelle Buildprozesse möglich, bei denen immer nur die notwendigen Schritte ausgeführt werden, was zu einer beträchtlichen Beschleunigung der Ausführung führen kann. Als Grundlage für ein solches Verständnis des Buildprozesses müssen die Ein- und Ausgabeartefakte identifiziert werden. Die zugehörige Modellierung des AMs wird in Abbildung 6.26 dargestellt.

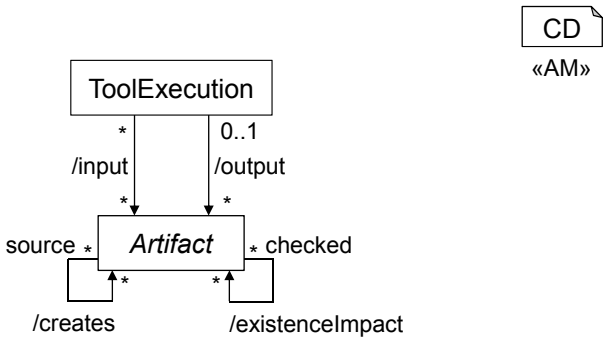

Abbildung 6.26.: Eingabe- und Ausgabeartefakte von Werkzeugausführungen.

Eine detaillierte Spezifikation der neu hinzugefügten Beziehungen wird in Listing 6.27 angegeben.

Durch die in Zeile 1 modellierte Beziehung wird die Menge aller Artefakte, die bei der entsprechenden Werkzeugausführung als Eingabe genutzt wurde, bereitgestellt. Zur Bestimmung dieser Menge werden alle `ArtifactRead`- und `ArtifactExistenceCheck`-Aktionen,

CD «AM»

```
1  association /input [*] ToolExecution -> Artifact [*];
2  association /read [*] ToolExecution -> Artifact [*];
3  association /existenceCheck [*] ToolExecution -> Artifact [*];
4
5  context ToolExecution inv:
6    read == {a in ArtifactRead | a in subaction**}.reads;
7
8  context ToolExecution inv:
9    existenceCheck ==
10     {a in ArtifactExistenceCheck | a in subaction**}.checks;
11
12 context ToolExecution inv:
13   input == read.addAll(existenceCheck);
14
15 association /output [0..1] ToolExecution -> Artifact [*];
16
17 context ToolExecution inv:
18   output ==
19     {a in ArtifactCreate | a in subaction**}.creates;
20
21 association /creates [*] Artifact (source) -> Artifact [*];
22 association /existenceImpact
23   [*] Artifact (checked) -> Artifact [*];
24
25 context Artifact inv:
26   creates == read.output;
27
28 context Artifact inv:
29   existenceImpact == existenceCheck.output;
30
31 context Artifact inv:
32   produces.containsAll(creates);
33
34 context Artifact inv:
35   produces.containsAll(existenceImpact);
```

Listing 6.27: AM: Abgeleitete Ein- und Ausgaben bei der Werkzeugausführung.

die als Unteraktionen der Werkzeugausführung ausgeführt werden, beachtet. Die Beziehung aus Zeile 15 modelliert entsprechend die Ausgabeartefakte von Werkzeugausführungen, zu deren Berechnung ausschließlich `ArtifactCreate`-Aktionen betrachtet werden. Die Assoziationen `creates` und `existenceImpact` modellieren, die Beziehungen zwischen eingelesenen Artefakten bzw. auf Existenz geprüften Artefakten und den erstellten Artefakten einer Werkzeugausführung. Sie stellen eine sehr konservative Abschätzung der Abhängigkeiten

zwischen Artefakten dar, da nicht feingranular entschieden wird, welche Eingabeartefakte tatsächlich eine Auswirkung auf bestimmte Ausgabeartefakte haben. Weiterhin sind beide Assoziationen als Spezialisierungen der `produces`-Beziehung modelliert.

6.5.4. Beteiligung von Java und Templatedateien an der Generierung

Da es unter ausschließlicher Verwendung der `produces`-Beziehung nicht möglich ist, feingranular zu entscheiden, welches Artefakt des Generators zur Erstellung eines generierten Artefakts des Produkts beigetragen hat, werden dem AM in diesem Abschnitt entsprechende Beziehungen für Java-Arefakte und Templates hinzugefügt. In Abbildung 6.28 wird eine Übersicht über die neu hinzugefügten Beziehungen gegeben.

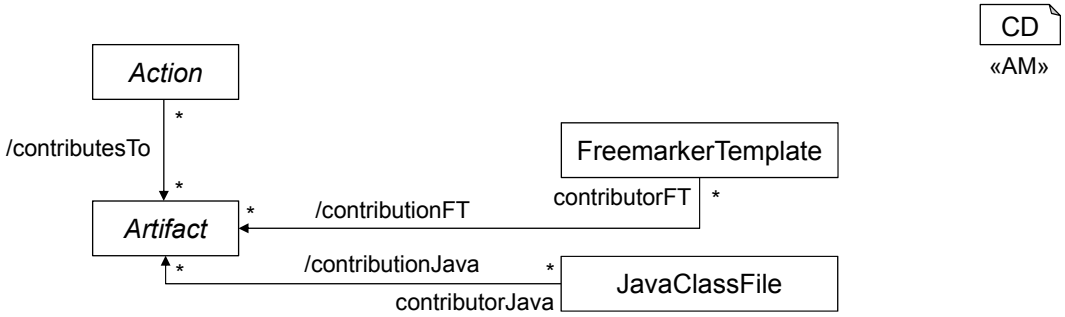

Abbildung 6.28.: Template- und Java-Artefakte tragen zur Erstellung von Artefakten bei.

Generatoren sind Werkzeuge und bestehen somit selbst aus einer Menge an Artefakten. Zusätzlich wird bei ihrer Ausführung eine variable Menge an Eingabeartefakten verwendet. Artefakte aus beiden Mengen können zur Generierung von Ausgabeartefakten beitragen. Für werkzeuginterne Artefakte wird dieser Beitrag durch entsprechende Aktionen hervorgerufen und in der `contributesTo`-Beziehung zusammengeführt. In Listing 6.29 wird die detaillierte Spezifikation dieser Beziehungen vorgenommen.

Um die `contributesTo`-Beziehung für eine Aktion berechnen zu können, muss die entsprechende Aktionsart bekannt sein. Darüber hinaus wird die Aktionshierarchie bei einigen Aktionsarten zu dieser Berechnung benötigt, wie es beispielsweise für `TemplateExecutions` und `JavaMethodCalls` der Fall ist. Bei den Aktionen zur Erstellung von Artefakten oder dem Schreiben von Strings kann hingegen direkt auf Assoziationen der entsprechende Aktionen zurückgegriffen werden. Die Berechnungsvorschriften der `contributesTo`-Beziehung für die konkreten Aktionsarten werden in den Zeilen 3 bis 24 von Listing 6.29 definiert.

Basierend auf der so berechneten `contributsTo`-Beziehung von Aktionen zu den erstellten Artefakten lassen sich nun auch Beziehungen von Template- und Java-Artefakten zu erstellten Artefakten wie in Listing 6.30 definieren.

CD
«AM»

```
1  association /contributesTo [*] Action -> Artifact [*];
2
3  context ToolExecution inv:
4    contributesTo == sub.contributesTo;
5
6  context ToolExecution inv:
7    contributesTo == output;
8
9  context TemplateExecution inv:
10   contributesTo == sub.contributesTo.add(writesTo);
11
12 context JavaMethodCall inv:
13   contributesTo == sub.contributesTo;
14
15 context ArtifactRead inv:
16   contributesTo == {};
17
18 context ArtifactExistenceCheck inv:
19   contributesTo == {};
20
21 context ArtifactCreate inv:
22   contributesTo == {creates};
23
24 context WriteString inv:
25   contributesTo == {writesTo};
```

Listing 6.29: AM: Aktionen tragen zur Generierung von Artefakten bei.

In den Zeilen 1 und 10 werden die entsprechenden Assoziationen für Java-Artefakte und Templates definiert. Die zugehörigen Berechnungsvorschriften sind in den Zeilen 4 und 13 definiert.

Die Beziehungen `contributionFT` and `contributionJava` helfen zum einen dabei, zu verstehen, welche Artefakte des Generators angepasst oder ersetzt werden müssen, wenn die Generierung für ein bestimmtes Ausgabeartefakt angepasst werden soll. Zum anderen wird aber auch die Auswirkung von Änderungen an Generatorartefakten auf generierte Ausgabeartefakte sichtbar, wodurch ungewollte Seiteneffekte bei Änderungen vermieden werden können. Neben den Berechnungsvorschriften für beide Beziehungen wird in den Zeilen 7 und 16 dargestellt, dass beide Beziehungen Spezialisierungen der `produces`-Beziehung sind.

Nachdem nun das AM für MontiCore-basierte Entwicklungsprojekte definiert wurde, werden in Abschnitt 6.6 beispielhafte Erweiterungen des AMs zu Analysezwecken vorgestellt. Diese demonstrieren, wie sich das AM zur artefaktbasierten Analyse einsetzen lässt. Konkrete Anwendungen zur Untersuchung realer Projekte folgen dann in Kapitel 10.

CD
«AM»

```
1  association /contributionFT
2    [*] FreeMarkerTemplate (contributorFT) -> Artifact [*];
3
4  context FreeMarkerTemplate inv:
5    contributionFT == templateExecution.contributesTo;
6
7  context FreeMarkerTemplate inv:
8    produces.containsAll(contributionFT);
9
10 association /contributionJava
11   [*] JavaClassFile (contributorJava) -> Artifact [*];
12
13 context JavaClassFile inv:
14   contributionJava == javaMethodCall.contributesTo;
15
16 context JavaClassFile inv:
17   produces.containsAll(contributionJava);
```

Listing 6.30: AM: Artefakte tragen zur Generierung anderer Artefakte bei.

6.6. Beispielhafte Analysespezifikationen

Das in diesem Kapitel vorgestellte AM beschreibt MontiCore-basierte Entwicklungsprojekte durch die Angabe der wichtigsten Konzepte und deren Beziehungen. Basierend auf diesem Modell können nun Erweiterungen zu Analysezwecken definiert werden, die auf Artefaktdaten ausgewertet werden können, die zum AM-passenden. Durch die Wahl einer sehr flexiblen Modellierungstechnik sind prinzipiell beliebig viele solcher Analysen denkbar. In diesem Abschnitt wird eine Auswahl beispielhafter Analysespezifikationen vorgestellt. In Kapitel 10 folgt später die Vorstellung konkreter Analysen von realen Projekten.

6.6.1. Architekturrekonstruktion

Bei der Architekturrekonstruktion (siehe Abschnitt 3.5) werden die Beziehungen zwischen Architektureinheiten des Systems rekonstruiert, welche hier durch Module (vgl. Abschnitt 5.2.2) repräsentiert werden. Module können dabei sowohl für die Werkzeuge des Projekts als auch für das zu entwickelnde Produkt definiert werden. Durch eine explizite Modellierung von Modulen und der Zuweisung von Artefakten des Projekts zu Modulen lässt sich eine abstrakte Projektansicht erstellen, die den Stakeholdern (*Architekten*, *Entwicklern* oder *Werkzeugentwicklern*) eine gute Übersicht über das MDD-Projekt ermöglicht. Module können dabei oftmals durch die vorhandenen Pakete oder Ordner identifiziert werden. Allerdings kann durch Architekturerosion (vgl. Abschnitt 3.5) diese Zuordnung verloren gehen. Außerdem ist es möglich, für ein

einzelnes Projekt verschiedene Architekturen zu definieren, die jeweils einen anderen Fokus besitzen. In [Lil16] werden beispielsweise die folgenden Arten von Architekturen betrachtet:

- Bei der *Technischen Schichtung* werden Module nach ihren technischen Aufgaben eingeteilt. Beispiele für technische Module sind Präsentations-, Applikations- oder Fachdomänenmodule.

- Bei der *Fachlichen Schichtung* repräsentieren Module die Elemente bzw. Prozesse der Anwendungsdomäne des Systems.

- Bei der Verwendung von *Mustersprachen* werden die Artefakte zu Modulen zusammengefasst, die im Projekt dieselbe Rolle einnehmen. Zum Beispiel könnten alle Artefakte, die eine `View` des Model-View-Controller-Musters [Fow02] realisieren in einem Modul zusammengefasst werden.

Darüber hinaus lassen sich im Allgemeinen verschiedene Arten von Modulen für eine Softwarearchitektur, wie etwa Komponenten, Schichten oder Schnittstellen, definieren. Das AM definiert eine einzelne, hierarchisch dekomponierbare Substruktur. Diese erlaubt die flexible Zuordnung von Artefakten und kann zur Definition verschiedener Architekturen desselben Projekts verwendet werden. Generell ist dabei zu empfehlen, dass die vorgegebenen Module möglichst gut durch eine entsprechende Strukturierung im Projekt widergespiegelt werden, da sich die Stakeholder so besser in dem Projekt zurechtfinden können. Die Beziehungen zwischen Modulen können, wie in Listing 6.31 dargestellt, basierend auf den zugewiesenen Artefakten abgeleitet werden.

Es wird zwischen zwei verschiedenen Arten von Modulbeziehungen, die basierend auf den zugewiesenen Artefakten und deren Beziehungen berechnet werden können, unterschieden. Die `reliesOnModule`-Beziehung gibt an, dass sich ein Modul auf ein anderes Modul bezieht, während die `contributionModule`-Beziehung die Beteiligung eines Moduls an der Erstellung eines anderen Moduls angibt. Zur Berechnung der Beziehungen wurden weitere abgeleitete Hilfsassoziationen definiert. Module sind hierarchisch dekomponierbar und jedes Modul kann eine Menge an Artefakten enthalten. Alle Submodule der Modulhierarchie werden in der Assoziation `containedModule` zusammengefasst, während die `containedArtifact`-Beziehung alle zugewiesenen Artefakte inklusive der Artefakte der Submodule enthält. Darüber hinaus können pro Modul die externen Artefakte berechnet werden, die mit Artefakten des Moduls oder einem seiner Submodule in Beziehung stehen, selbst aber keinem dieser Module zugeordnet sind. Dabei werden die Beziehungen je nach Art der zugrundeliegenden Artefaktbeziehungen durch die `externalReferredArtifact`- bzw. die `externalProducedArtifact`-Beziehung in der gezeigten Erweiterung des AMs repräsentiert.

Mit Hilfe der so definierten Assoziationen lassen sich nun die Berechnungen der `reliesOnModule`- sowie der `contributionModule`-Beziehungen spezifizieren, was der Berechnung der Ist-Architektur (vgl. Abschnitt 3.5) gleichkommt.

CD
«AM4A»

```
1  association /containedModule [*] Module -> Module [*];
2
3  context Module inv:
4    containedModule == {this}.addAll(subModule**);
5
6  association /containedArtifact [*] Module -> Artifact [*];
7
8  context Module inv:
9    containedArtifact == containedModule.artifact;
10
11 association /externalReferredArtifact
12   [*] Module -> Artifact [*];
13 association /externalProducedArtifact
14   [*] Module -> Artifact [*];
15
16 context Module inv:
17   externalReferredArtifact ==
18     containedArtifact.refersTo.removeAll(containedArtifact);
19
20 context Module inv:
21   externalProducedArtifact ==
22     containedArtifact.produces.removeAll(containedArtifact);
23
24 association /reliesOnModule [*] Module -> Module [*];
25 association /contributionModule [*] Module -> Module [*];
26
27 context Module inv:
28   reliesOnModule == externalReferredArtifact.module;
29
30 context Module inv:
31   contributionModule == externalProducedArtifact.module;
```

Listing 6.31: Erweiterung des AMs zu Analysezwecken: Architekturrekonstruktion.

6.6.2. Differenzarchitektur

Aufbauend auf der in Abschnitt 6.6.1 rekonstruierten Ist-Architektur lässt sich mit Hilfe von Erweiterungen des AMs ebenfalls die Differenzarchitektur zwischen Soll- und Ist-Architektur berechnen. Die erste benötigte Erweiterung zu Analysezwecken ist in Listing 6.32 dargestellt und ermöglicht die Angabe der Soll-Architektur.

Mit Hilfe der beiden Assoziationen und `intendedReference` `intendedContribution` lassen sich die vorgesehen Beziehungen zwischen Modulen spezifizieren. Diese können anschließend dazu genutzt werden, die Unterschiede zwischen Soll- und Ist-Architektur zu berechnen.

```
association intendedReference [*] Module -> Module [*];
association intendedContribution [*] Module -> Module [*];
```
CD «AM4A»

Listing 6.32: Erweiterung des AMs zu Analysezwecken: Soll-Architektur.

Die in Listing 6.33 dargestellte zweite Erweiterung des AMs zu Analysezwecken ermöglicht die Darstellung der Differenzarchitektur und spezifiziert deren Berechnung durch OCL-Ausdrücke.

```
association /unintendedReference [*] Module -> Module [*];
association /unintendedContribution
  [*] Module -> Module [*];

association /missingReference [*] Module -> Module [*];
association /missingContribution [*] Module -> Module [*];

context Module inv:
  unintendedReference ==
    reliesOnModule.removeAll(intendedReference);

context Module inv:
  unintendedContribution ==
    contributionModule.removeAll(intendedContribution);

context Module inv:
  missingReference ==
    intendedReference.removeAll(reliesOnModule);

context Module inv:
  missingContribution ==
    intendedContribution.removeAll(contributionModule);
```
CD «AM4A»

Listing 6.33: Erweiterung des AMs zu Analysezwecken: Differenzarchitektur.

Um die Ergebnisse der Berechnungen erfassen zu können, werden vier abgeleitete Assoziationen benötigt. Pro zu vergleichender Beziehungsart werden jeweils zwei Assoziationen spezifiziert, wovon eine die unerwarteten, zusätzlichen Beziehungen der Ist-Architektur im Vergleich zur Soll-Architektur und die andere die fehlenden, erwarteten Beziehungen enthält. Berechnen lassen sich die unerwarteten, zusätzlichen Beziehungen in dem aus der Menge der bestehenden Beziehungen die erwarteten Beziehungen entfernt werden. Im Gegensatz dazu werden zur Berechnung von fehlenden, erwarteten Beziehungen die bestehenden Beziehungen aus der Menge der erwarteten Beziehungen entfernt.

6.6.3. Indirekte Templatebeziehungen

Bei der Nutzung templatebasierter Generierung tritt oftmals das folgende Problem auf. Große Anteile von Templates werden direkt in der Sprache des generierten Zielsystems geschrieben und nur kleine Anteile werden durch Ausdrücke der Templatesprache selbst dargestellt. Somit sind Templates oft nicht direkt auf andere Templates angewiesen. Die durch verschiedene Templates generierten Artefakte können hingegen schon aufeinander angewiesen sein. In solchen Fällen kann die Änderung an einem Template leicht zu Fehlern im generierten Quellcode führen. Aus diesem Grund ist die Kenntnis über diese Art von indirekter Beziehung zwischen Templates nützlich, um beispielsweise die Auswirkung von Änderungen abschätzen zu können. In Listing 6.34 wird eine entsprechende Erweiterung des AMs zu Analysezwecken angegeben.

CD
«AM4A»

```
1  association /contributionsRelyOn
2    [*] FreeMarkerTemplate -> FreeMarkerTemplate [*];
3
4  context FreeMarkerTemplate inv:
5    contributionsRelyOn == contributionFT.contributorFT
6      .addAll(contributionFT.refersTo.contributorFT);
```

Listing 6.34: Erweiterung des AMs zu Analysezwecken: Indirekte Templatebeziehungen.

Mit Hilfe der neu hinzugefügten `contributionsRelyOn`-Beziehung zwischen FreeMarker-Templates lassen sich die Ergebnisse einer solchen Analyse festhalten. Zwei Templateartefakte werden dort genau dann verlinkt, wenn entweder beide Templates zur Generierung desselben Artefakts beitragen oder die von den Templates generierten Artefakte aufeinander angewiesen sind. Durch die dargestellte Berechnungsvorschrift zu Analysezwecken wird auch eine Zuordnung von Templates zu sich selbst vorgenommen, was bei einer Interpretation des Ergebnisses berücksichtigt werden muss.

Die Analyse gibt Hinweise darauf, dass bei der Änderung eines Templateartefakts die Änderung an den verlinkten Templates notwendig sein könnte. Eine genaue Aussage darüber, an welcher Stelle Änderungen wirklich notwendig sind, ist hingegen nicht möglich. Dafür gibt es zwei Gründe: (1) Da die Analyse von der letzten Ausführung des Buildprozesses abhängt, kann nicht sichergestellt werden, dass sich z.B. bei einer Änderung der Eingabe nicht noch weitere solcher indirekten Beziehungen ergeben. Somit können die durch die Analyse berechneten Beziehungen nur als vollständig in Bezug auf die letzte Ausführung des Buildprozesses angenommen werden. (2) Das Vorhandensein einer Beziehung garantiert nicht, dass eine Änderung an dem verlinkten Template wirklich notwendig ist. Dies hängt sowohl von dem Inhalt der Templatedateien als auch von der Art der Änderung ab.

6.6.4. Ungenutzte Importe

Code Smells [FB99] reduzieren die Lesbarkeit des Quellcodes und können dadurch die Wartungskosten erhöhen. Eine Form von Code Smells sind ungenutzte Importe. Für Java können moderne IDEs ungenutzte Importe bereits automatisch erkennen und entfernen. Basierend auf dem AM lassen sich ungenutzte Importe auch zwischen anderen Artefaktarten aufdecken. Dazu können die in Listing 6.35 modellierten Erweiterungen des AMs zu Analysezwecken verwendet werden.

```
1  association /typeUsage [*] JavaArtifact -> JavaArtifact [*];   CD
2                                                                  «AM4A»
3  association /unusedImports
4    [*] JavaSourceFile -> JavaArtifact [*];
5
6  context JavaSourceFile inv:
7    typeUsage == defines.reliesOn.javaArtifact.remove(this);
8
9  context JavaSourceFile inv:
10   unusedImports == imports.removeAll(typeUsage);
11
12 association /typeUsage [*] CDModelFile -> Artifact [*];
13
14 association /unusedImports [*] CDModelFile -> Artifact [*];
15
16 context CDModelFile inv:
17   typeUsage == defines.reliesOn.javaArtifact
18     .addAll(defines.reliesOn.cDModelFile).remove(this);
19
20 context CDModelFile inv:
21   unusedImports == imports.removeAll(typeUsage);
```

Listing 6.35: Erweiterung des AMs zu Analysezwecken: Ungenutzte Importe.

Bei ungenutzten, importierten Artefakten werden die von den importierten Artefakten bereitgestellten Typen nicht von den importierenden Artefakten genutzt. Diese Nutzungsbeziehung von Typen lässt sich auch auf der Artefaktebene modellieren, was durch die `typeUsage`-Beziehung realisiert wurde. Mit Hilfe dieser Beziehung lassen sich dann die ungenutzten `imports`-Beziehungen von Java-Quellcodedateien (Zeile 3) berechnen. Dazu werden aus der Menge der importierten Artefakte diejenigen entfernt, die durch eine Typnutzungsbeziehung verbunden sind. Entsprechende Beziehungen können aber auch für andere Artefaktarten spezifiziert werden. In Listing 6.35 wurde dies für CD-Modelldateien vorgenommen. Importierte, aber ungenutzte Dateien werden hier durch die in (Zeile 14) dargestellte `unusedImports`-Beziehung repräsentiert. Wie in Abschnitt 6.3 definiert, können CD-Modelldateien verschiedene Arten von Artefakten importieren. Diese Arten wurden nicht weiter eingeschränkt, weshalb als

Ziel der Assoziation die `Artifact`-Klasse des AMs gewählt wurde. Trotzdem müssen bei der Definition der `typeUsage`-Beziehung für CD-Modelldateien (siehe Zeile 16) die relevanten Artefaktarten explizit angegeben werden.

6.6.5. Inkrementelle Ausführung von Werkzeugketten

Die Ausführungszeit von Werkzeugketten sollte möglichst kurz sein. Anderenfalls verlieren *Entwickler* ihren Entwicklungsfluss [Csi96], was zu einer eingeschränkten Produktivität führt. Ein effektiver Weg die Ausführungszeit von Werkzeugketten zu reduzieren ist, ausschließlich die relevanten Teile der Werkzeugkette auszuführen. Diese Teile lassen sich durch die Struktur der Werkzeugkette und durch Betrachtung der letzten Änderungen identifizieren. Dazu wurde in Listing 6.36 das boolesche Attribut `toRepeat` zur AM-Klasse `ToolExecution` hinzugefügt. Anders als bei den vorherigen Erweiterungen des AMs zu Analysezwecken werden an dieser Stelle auch bestehende Klassen des AMs erweitert.

CD
«AM4A»

```
1  abstract class Artifact {
2    //...
3    boolean removed;
4    boolean isHandcoded;
5    /boolean toBeRecreated;
6  }
7
8  class ToolExecution extends Action {
9    /boolean toRepeat;
10 }
11
12 context Artifact inv:
13   isHandcoded implies !toBeRecreated;
```

Listing 6.36: Erweiterung bestehender AM Klassen als Vorbereitung zur Analyse der neu auszuführenden Teile von Werkzeugketten.

Neben der `ToolExecution`-Klasse wird auch die `Artifact`-Klasse um drei Attribute erweitert. Durch das boolesche Attribut `removed` wird angezeigt, dass ein entsprechendes Artefakt seit der letzten Beobachtung gelöscht worden ist. Um dies in den Artefaktdaten angeben zu können, kann beispielsweise das Dateisystem beobachtet werden. Da gelöschte Artefakte zum gewählten Beobachtungszeitpunkt (siehe Abschnitt 4.4) nicht mehr existieren, sind sie als virtuell anzusehen. Zusammen mit dem bereits im AM existierenden `modified`-Attribut (siehe Listing 5.3) lassen sich Modifikationen und das Entfernen von Artefakten festhalten. Auch die Neuerstellung von Artefakten seit dem letzten Durchlauf der Werkzeugkette lässt sich mit Hilfe des `modified`-Attributs bestimmen.

Als weiteres Attribut beschreibt das `isHandcoded`-Attribut, ob ein Artefakt generiert oder händisch erstellt wurde. Wie in Abschnitt 2.2 beschrieben, wird von der Verwendung von Mischformen (teilweise generierten Artefakten) abgesehen und auf die nachträgliche Veränderung von generierten Artefakten wird ebenfalls verzichtet. Diese Einschränkungen sind notwendig, um einen automatischen und wiederholbaren Buildprozesses im Projekt nutzen zu können. Ob ein Artefakt handgeschrieben ist oder automatisch erstellt wurde, ließe sich durch Beobachtung der `CreateActions` (siehe Abschnitt 6.5.2) und Angabe einer Berechnungsvorschrift automatisch berechnen. Stattdessen wurde das Attribut hier einfachheitshalber als nichtabgeleitetes Attribut modelliert, das direkt bei der Extraktion von Artefaktdaten mitgeliefert wird. Durch das abgeleitete Attribut `toBeRecreated` wird angegeben, ob ein automatisch erstelltes Artefakt bei der nächsten Ausführung der Werkzeugkette neu erstellt werden muss.

Für jede `ToolExecution` ist bereits bekannt, welche Artefakte bei Ausführung als Eingaben dienten und welche Artefakte als Ausgebe produziert wurden. Diese Informationen sind darüber hinaus bereits zur Berechnung der `creates`-Beziehung verwendet worden, sodass die `source`-Rolle dieser abgeleiteten Assoziation für die durchzuführende Analyse verwendet werden kann. Dadurch lässt sich die Berechnung des `toBeRecreated`-Attributs, wie in Listing 6.37 dargestellt, spezifizieren.

CD «AM4A»

```
1  context Artifact inv:
2    toBeRecreated == !isHandcoded && (
3      removed ||
4      modified < max(source.modified) ||
5      {a in source | a.toBeRecreated}.size > 0);
```

Listing 6.37: Erweiterung des AMs zu Analysezwecken: Neu zu erstellende Artefakte.

Ein Artefakt muss neu erstellt werden, wenn es generiert und anschließend entweder gelöscht (Zeile 3) wurde oder das Datum seiner letzten Änderung älter als das Datum eines der `source`-Artefakte ist (Zeile 4). Die Notwendigkeit zur Neuerstellung besteht auch dann, wenn ein entsprechend verlinktes Artefakt selbst neu erstellt werden muss (Zeile 5).

In Listing 6.38 wird darauf aufbauend die Berechnung des `toRepeat`-Attributs angegeben, wodurch sich die Notwendigkeit von Neuausführungen bestimmter Teile der Werkzeugkette bestimmen lässt. Dies ist der Fall, wenn eines der Ausgabeartefakte neu erstellt werden muss.

CD «AM4A»

```
1  context ToolExecution e, Artifact a in e.output inv:
2    a.toBeRecreated implies e.toRepeat;
```

Listing 6.38: Erweiterung des AMs zu Analysezwecken: Neu auszuführende Teile der Werkzeugkette.

Der Vollständigkeit halber bleibt zu erwähnen, dass bei der spezifizierten Analyse das Hinzufügen neuer Eingaben nicht mit betrachtet wird. Ein neu hinzugefügtes Eingabeartefakt führt aber auch zu einer Neuausführung der entsprechenden Werkzeuge. Für eine vollständige Analyse muss dieser Fakt berücksichtigt und die gezeigte Analyse ergänzt werden.

6.7. Verwandte Ansätze

In diesem Abschnitt werden mit dem AM verwandte Ansätze beschrieben. Diese Ansätze modellieren ebenfalls Artefakte und deren Beziehungen zueinander. Teilweise werden dabei auch Werkzeuge mit betrachtet.

6.7.1. Megamodelle

Megamodelle [BJV04] (oder auch Macromodelle [SME09]) stellen eine globale (oder auch makroskopische) Sicht auf MDD-Projekte bereit. Modelle des MDD-Projekts werden in Megamodellen selbst als Modellelemente repräsentiert, während Links des Megamodells Beziehungen zwischen den Modellen repräsentieren [SKCM16]. Die Verwendung von Megamodellen wird auch als *modeling-in-the-large* [BJRV05] bezeichnet. Dabei bedeutet makroskopische Sichtweise, dass Modelle als Ganzes betrachtet werden und von den inneren Elementen abstrahiert wird. Auch Beziehungen werden vor allem zwischen den Modellen des MDD-Projekts betrachtet. Trotzdem existieren in diesem Gebiet auch Ansätze, die ebenfalls innere Elemente und deren Beziehungen mit berücksichtigen [SME09, HSG11, Sei13]. Somit ist der Betrachtungsumfang von Megamodellen und dem in diesem Kapitel vorgestellten AM zumindest ähnlich. Auch im AM wurde eine globale Sicht auf die Artefakte eingenommen, zu denen auch Modellartefakte gehören. Artefaktinterne Elemente wurden ebenfalls an einigen Stellen betrachtet, wenn auch in einem geringen Detailgrad.

In Megamodellen werden neben Modellen auch Werkzeuge, Services, statische Kenntnisbeziehungen zwischen Modellen und Metamodellen sowie dynamische Ablaufbeziehungen durch Modelltransformationen betrachtet [BJV04]. Der Einsatzzweck von Megamodellen ist dabei unterschiedlich. Während in [BJV04] Megamodelle automatisch bei Auftreten eines relevanten Events aktualisiert werden, um eine Übersicht über den aktuellen Projektzustand bereitzustellen, werden in [FN05, Fav06] Megamodelle dazu genutzt, um MDE-Muster zu definieren. In [Sei13] werden Megamodelle dazu verwendet, um Traces zu speichern, heterogene Modelloperationen zu spezifizieren und diese auszuführen. [HSG11] gibt eine Übersicht über die verschiedenen Anwendungen von Megamodellen. Im Gegensatz zu den oben diskutierten unterschiedlichen Verwendungen von Megamodellen hat das in diesem Kapitel vorgestellte AM vor allem die Analyse zugehöriger Artefaktdaten zum Ziel. Dazu können Analysen auf einem AM spezifizieren, zugehörige Artefaktdaten automatisiert aus dem Projekt extrahiert und die spezifizierten Analysen automatisiert durchführen werden (vgl. Kapitel 7 und Kapitel 8).

Darüber hinaus kann das AM zu Dokumentations- und Kommunikationszwecken verwendet werden.

In [HSG11] werden die Megamodellansätze verglichen und Gemeinsamkeiten herausgearbeitet. Alle Megamodellansätze haben gemeinsam, dass sie eine globale Sicht auf Modelle (oder Artefakte) darstellen und von bestimmten internen Sachverhalten abstrahieren. Zusätzlich ist ein Megamodell selbst ein Modell. Eine verallgemeinerte Definition des Begriffs Megamodell wird in [HSG11] angegeben: *„a model that contains models and relations between them.“*. Darüber hinaus wurde aus den Ansätzen ein Kernmetamodell für Megamodelle extrahiert, welches für bestimmte Anwendungszwecke erweitert werden kann [HSG11]. Megamodelle sind weiterhin Bestandteil der aktuellen Forschung, was beispielsweise die Arbeiten [SPBS15, SKCM16, TFB+17] zeigen.

Das in diesem Kapitel beschriebene AM kann als Metamodell für zugehörige Megamodelle angesehen werden, während Artefaktdaten als Pendant zu einem Megamodell ansehen werden können. Artefaktdaten werden in einer Modellierungssprache beschrieben, stellen eine makroskopische Sicht auf ein MDD-Projekt dar und können Modelle des Projekts als Modellelemente enthalten. Mit dem in Abschnitt 5.2 vorgestellten AM-Kern existiert weiterhin ein wiederverwendbarer Teil, ähnlich zu dem Kernmetamodell für Megamodelle aus [HSG11]. Die Notwendigkeit, projektspezifische Anpassungen vorzunehmen, gilt ebenfalls sowohl für Megamodelle [BJV04] als auch für das AM (siehe Abschnitt 4.4). Da ein solches Metamodell sehr allgemein gehalten sein muss, um die verschiedenen Ansätze zu unterstützen, muss bei der Erstellung von projektspezifischen Varianten ein Großteil der konzeptionellen Modellierung noch erbracht werden. Aus diesem Grund kann das in diesem Kapitel beschriebene AM unabhängig von der gewählten Modellierungstechnik als Beitrag zu diesem Gebiet angesehen werden.

Trotz der oben beschriebenen Gemeinsamkeiten mit Megamodellen werden in dieser Arbeit andere Begrifflichkeiten verwendet. Zum einen wird eine andere Sichtweise des Modellbegriffs angenommen. Während in [Béz05] alles als Modell und in [Fav06] alles als System angesehen wird, wird in dieser Arbeit zwischen Modellen, Artefakten anderer Art und weiteren Elementen strikt unterschieden. Ein weiterer Unterschied ist, dass MDD-Werkzeuge nicht als Modelle oder atomare Modellelemente eines Megamodells angesehen werden, sondern eine Whiteboxsicht auf Werkzeuge angenommen wird. Dadurch lassen sich die interne Struktur von Werkzeugen und deren Ausführung im Detail mitbetrachten, was bei Analysen Aussagen über die Zusammenhänge von MDD-Werkzeugen, deren Artefakten sowie den Eingabe- und Ausgabeartefakten ermöglicht. Dies ist vor allem dann wichtig, wenn MDD-Werkzeuge agil und parallel zum Zielprodukt weiterentwickelt oder durch Konfigurationen angepasst werden. Darüber hinaus gibt es Buildprozesse, bei denen Teile der MDD-Werkzeuge selbst generiert werden, wodurch ein mehrstufiger Generierungsprozess entsteht. Unter der Annahme, dass sich die eingesetzten MDD-Werkzeuge in den verschiedenen Schritten des Buildprozesses stark konfigurieren lassen, entsteht eine Komplexität, die sich jedoch mit dem Ansatz dieser Arbeit beherrschen lässt.

6.7.2. Artefaktorientierung und Artefaktmodelle

Der Begriff *Artefaktorientierung* wird in [FPKB10, MFP15] im Kontext der Anforderungsanalyse dazu verwendet, sich von prozessorientierten Vorgehensweisen abzugrenzen. Prozessorientierte Vorgehensweisen der Anforderungsanalyse geben vor, welche Prozessschritte in welcher Reihenfolge durchzuführen sind, um Artefakte aus dem Bereich der Anforderungsanalyse zu erstellen (an dieser Stelle ist der Begriff Artefakt mit dem Ergebnis eines Prozessschritts gleichzusetzen). Demgegenüber steht der artefaktorientierte Ansatz, welcher keine feste Reihenfolge oder Vorgehensweisen für Prozessschritte definiert. Stattdessen gibt dieser Ansatz vor, welche Artefaktinhalte zu erstellen sind und wie diese Inhalte zusammenhängen. Somit liegt der Fokus genau wie bei dem in diesem Kapitel vorgestellten AM auf den Artefakten und nicht auf dem Vorgehen, um zu diesen zu gelangen.

Die zu erstellenden Artefaktinhalte und deren Beziehungen werden in dem oben beschriebenen Ansatz explizit modelliert. Dazu wird in [MFP15] ein Referenzartefaktmodell verwendet, welches sich auf verschiedenen Ebenen an die Bedürfnisse von Organisationen oder einzelnen Projekten zuschneiden lässt. Das Modell unterstützt drei verschiedene Artefaktarten: *Kontextspezifikation*, *Anforderungsspezifikation* und *Systemspezifikation*. Artefakte enthalten in diesem Fall jedoch weitere Inhalte (*content items*) deren Struktur und Beziehungen vorgegeben sind. In [FPKB10] wird außerdem ein zugehöriges Metamodell für solche Artefaktmodelle vorgestellt. Das Referenzartefaktmodell lässt sich mit dem in dieser Arbeit verwendeten Artefaktmodell vergleichen. In beiden Fällen werden Artefakt- und Beziehungsarten definiert, die dann projektspezifische Instanzen haben können. Ansonsten gibt es aber auch einige Unterschiede. Der oben genannte Ansatz hat die Strukturierung von Artefaktinhalten zum Ziel, unterstützt hauptsächlich die Anforderungserhebung und modelliert eine Vorgabe für die manuelle Erstellung von Artefakten. Demgegenüber wird in dieser Arbeit von Artefaktinhalten abstrahiert, auf den automatisierten MDD-Prozess fokussiert und eine automatische Extraktion von Artefaktdaten zu Analysezwecken angestrebt (siehe Abschnitt 8.2).

In [Bro13, Bro18] wird ebenfalls ein Artefaktmodell für Softwareentwicklungsprojekte vorgestellt. Das Modell unterscheidet zwischen den Artefaktarten *Anforderungen*, *Spezifikation* und *Architektur*. Artefakte sind benannte Elemente, deren Inhalt entweder Daten oder logische Aussagen über die Daten sind. Dabei wird die Modellierung durch Angabe aussagenlogischer Formeln vorgenommen. Mit dieser Art der Modellierung ist es möglich, mathematische Definitionen für Verfeinerung und hierarchische Dekomposition anzugeben, wodurch sich u.a. die Konsistenz zwischen den modellierten Artefakten nachweisen lässt. Das in diesem Kapitel vorgestellte AM unterscheidet sich von diesem Ansatz zum einen dadurch, dass eine andere Modellierungstechnik verwendet wird. Prüfbedingung zur Spezifikation von Konsistenzregeln werden per hier OCL spezifiziert. Außerdem steht die Konsistenz zwischen Artefakten aus verschiedenen Phasen des Entwicklungsprozesses nicht im Fokus der Anwendung. Stattdessen modelliert das AM einen automatisch durchführbaren MDD-Buildprozess, bei dem die Konsistenz zwischen Artefakten verschiedener Entwicklungsschritte durch automatische Trans-

formationen sichergestellt wird. Darüber hinaus wird feingranularer zwischen verschiedenen Artefaktarten der Entwicklungsphase unterschieden.

In [FBB08] wird ein Artefaktmodell für Projekte im Kontext von Enterprise Architekturen definiert. Der Projektbegriff umfasst dort, anders als in dieser Arbeit (vgl. Abschnitt 2.2), sowohl Business Design- als auch IT-Komponenten. Neben den zu erstellenden Artefakten werden im Gegensatz zu dem Umfang des AMs auch die zugehörigen Aktivitäten und Rollen modelliert. Somit unterscheidet sich der Betrachtungsumfang der beiden Ansätze signifikant.

In [KFG13] wird, basierend auf einer Literaturrecherche, ein generisches Artefaktmodell für agile Softwareentwicklungsprojekte vorgestellt. Dabei wurden verschiedene agile Vorgehensweisen (u.a. XP [Bec04] und Scrum [SB02]) untersucht, die dort vorhandenen Artefaktarten und deren Beziehungen analysiert und als Basis zur Erstellung eines generischen Artefaktmodells verwendet. Im Gegensatz zu dem in diesem Kapitel beschriebenen AM beschränkt sich das generische Modell nicht auf Entwicklungsartefakte, sondern betrachtet auch Artefaktarten aus dem Bereich Projektmanagement, wie etwa *Backlog*, *Burn-Down* oder *Release plan*. Dafür wird jedoch nicht zwischen den verschiedenen Artefaktarten der entwickelten Produkte unterschieden und Werkzeuge bei der Softwareentwicklung außen vor gelassen. In [FKSJ14] wird das generische Artefaktmodell erweitert und im Kontext eines Projekts um projektspezifische Anteile ergänzt.

Kapitel 7.

Methodik zur artefaktbasierten Analyse modellgetriebener Softwareentwicklungsprojekte

In diesem Kapitel wird die Methodik zur artefaktbasierten Analyse modellgetriebener Softwareentwicklungsprojekte vorgestellt. Abbildung 7.1 zeigt die drei Schritte der Gesamtmethodik.

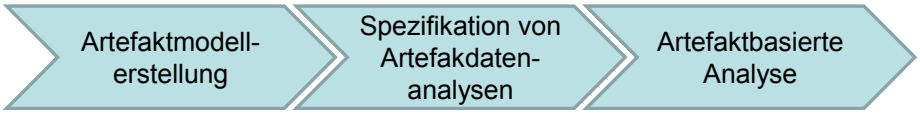

Abbildung 7.1.: Die Schritte der Gesamtmethodik zur artefaktbasierten Analyse.

Im ersten Schritt wird ein projektspezifisches AM erstellt. Anschließend werden unter Verwendung des projektspezifischen AMs Artefaktdatenanalysen spezifiziert. Einige Beispiele für solche Analysen wurden bereits in Abschnitt 6.6 vorgestellt. Das Vorgehen zu diesen beiden Schritten der Gesamtmethodik wurde basierend auf den Erfahrungen bei der Spezifikation des in Kapitel 6 vorgestellten AMs und dessen Anwendung zu Analysezwecken definiert. Der dritte Schritt, die arefaktbasierte Analyse, ist in mehrere Teilschritte unterteilt. Diese sind in Abbildung 7.2 dargestellt.

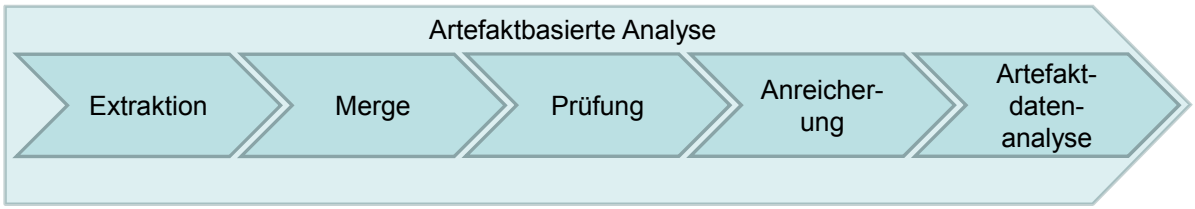

Abbildung 7.2.: Übersicht über die Teilschritte der artefaktbasierten Analyse.

Bei der Durchführung von artefaktbasierten Analysen werden zunächst die relevanten Daten des Projekts extrahiert. Sollten die Daten in verschiedenen Dateien vorliegen, findet anschließend ein *Merge* der Daten statt. Daraufhin wird die gesamte Datenmenge auf ihre Konformität

zu dem genutzten AM geprüft. Im darauffolgenden Schritt werden die Daten basierend auf der Spezifikation des AMs angereichert, sodass auch die abgeleiteten Eigenschaften für den letzten Schritt, der Artefaktdatenanalyse, zur Verfügung stehen. Ein verwandter Ansatz wird in [Leh15] vorgestellt, der auf die Durchführung einer Change-Impact-Analyse in heterogenen Artefaktlandschaften im Softwareentwicklungsprozess spezialisiert ist. Die dort beschriebenen Schritte sind die Spezifikation von Extraktionsregeln zur automatisierten Extraktion von Artefaktbeziehungen und die Angabe von Regeln, welche die Auswirkungen von Änderungen definieren. Bei der Angabe der Regeln kann auf eine Taxonomie von Abhängigkeitstypen und eine fixierte Menge von Änderungsoperationen zurückgegriffen werden. Im Gegensatz dazu beschränkt sich der hier vorgestellte Ansatz nicht auf Change-Impact-Analysen. Er erlaubt es außerdem, neue Artefakt- und Beziehungsarten hinzuzufügen. Die einzelnen Schritte der Methodik werden im Rest des Kapitels im Detail erläutert. Ein weiterer verwandter Ansatz ist Q-ImPrESS [BTR08, BHT+10]. Dieser zielt darauf ab, Softwareevolution zu unterstützen, indem verschiedene Designänderungen in Hinblick auf die Erfüllung von nichtfunktionalen Anforderungen wie Performanz, Zuverlässigkeit und Wartbarkeit analysiert und verglichen werden. Dazu wird in einem ersten Schritt eine Extraktion der Softwarearchitektur durchgeführt, auf deren Basis dann Modelltransformationen für spezielle Analysen ausgeführt werden. Somit ist die Vorgehensweise, um zu den Analyseergebnissen zu gelangen, verwandt mit der hier vorgestellten Gesamtmethodik. Der größte Unterschied besteht jedoch darin, dass die Analysen in dieser Arbeit auf Artefaktbeziehungen in MDD-Projekten und nicht auf die Erfüllung nichtfunktionaler Anforderungen des Zielprodukts fokussieren. In [GTB18] wird ein ähnlicher Ansatz zur Analyse von Modelltransformationen skizziert. Dieser zielt auf die Analyse und Optimierung der Performanz von Modelltransformationen ab.

7.1. Methodik zur Erstellung projektspezifischer Artefaktmodelle

Der erste Schritt der Methodik sieht die Modellierung eines AMs vor. Dadurch wird zum einen der Betrachtungsumfang für projektspezifische Analysen festgelegt. Zum anderen präzisiert ein solches Modell die Beziehungen zwischen den Artefakten und die erwarteten Ausgangsbedingungen vor den Analysen. Darüber hinaus lassen sich durch die Verwendung der Sprachen CD und OCL (siehe Abschnitt 2.4.1) MDD-Werkzeuge zur Durchführung der Analysen nutzen. Das in dieser Arbeit vorgestellte AM (siehe Kapitel 6) kann für MontiCore-basierte MDD-Projekte, die eine CD-zu-Java-Transformation durchführen, direkt wiederverwendet werden. Für Projekte mit anderen Charakteristiken kann es aber notwendig sein, ein neues AM zu erstellen oder Änderungen an einem bestehenden AM vorzunehmen. An dieser Stelle lassen sich Techniken für das Modellmanagement einsetzen. Insbesondere existieren bereits Techniken zur Verwaltung und Versionierung von Modellen [KH10, TELW14], die jedoch nicht im Fokus dieser Arbeit stehen. Auf die Zusammenführung von Modellen bzw. Modellteilen wird hingegen in Abschnitt 7.4 noch genauer eingegangen.

Muss für das zu untersuchende Projekt ein neues AM erstellt oder ein existierendes AM angepasst werden, so kann nach der hier beschriebene Methodik vorgegangen werden. Damit unterstützt der Ansatz projektspezifische Anpassungen mit denen sich auch, wie in Anforderung *MRQ-3.2* gefordert, der Abstraktionsgrad festlegen lässt. Bei der Neuerstellung eines projektspezifischen AMs sollte der AM-Kern (siehe Abschnitt 5.2) wiederverwendet und wenn möglich auch Teile existierender AMe genutzt werden. Ein ähnlicher Ansatz wird in [HSG11] vorgestellt. Dort wird der Kern eines Metamodells für Megamodelle um spezialisierte Modell- und Beziehungsarten erweitert. Zum Hinzufügen fehlender Elemente sind die folgenden Schritte vorgesehen:

1. Elemente modellieren.
2. Vererbungen hinzufügen.
3. Eigenschaften hinzufügen.
4. Ableitbare Eigenschaften spezifizieren.
5. Assoziationsspezialisierungen behandeln.
6. Zusätzliche Bedingungen spezifizieren.

Bei der Durchführung der Schritte ist eine iterative, agile Herangehensweise vorgesehen, d.h., die jeweiligen Schritte müssen nicht direkt für alle Elemente durchgeführt werden. Stattdessen können zunächst nur einige der benötigten Elemente hinzugefügt werden, wodurch sich erste Analysen bereits sehr früh spezifizieren (siehe Abschnitt 7.2) und durchführen (siehe Abschnitt 7.6) lassen, ohne zuerst alle potentiell relevanten Elemente und Eigenschaften des AMs modellieren zu müssen.

Zunächst müssen in Schritt (1) die relevanten Elemente identifiziert und als Klassen des AMs modelliert werden. Da der Fokus bei der artefaktbasierten Analyse auf den Artefakten und ihren Beziehungen liegt sollte mit diesen Elementen begonnen werden. Wie in Kapitel 6 gezeigt wurde, ist auch die Modellierung anderer Elemente (Module, Typen, etc.) möglich.

Bilden neu hinzugefügte Elemente Spezialisierungen von existierenden Oberklassen des vorhandenen Teils des AMs, so werden in Schritt (2) an diesen Stellen Vererbungsbeziehungen hinzugefügt. Dies trägt zur Erfüllung von Anforderung *MRQ-3.2* bei, da sich dadurch verschiedene Abstraktionsstufen in dem AM vereinen lassen. Dadurch können, zusätzlich zu sehr spezifischen Analysen, auch Analysen auf abstrakten Elementen durchgeführt werden.

Um die Elemente vollständig zu spezifizieren, müssen ihre Eigenschaften in Schritt (3) modelliert werden. Dazu werden dem AM Attribute, Assoziationen und Kompositionen hinzugefügt. Welche Eigenschaften nachher für Analysen von Interesse sind, kann nicht immer vorab bestimmt werden.

In Schritt (4) muss nun definiert werden, welche Eigenschaften abgeleitet werden können. Sowohl Attribute als auch Assoziationen und Kompositionen des AMs können abgeleitet sein.

Dies bedeutet, dass sich der Wert des zugehörigen Elements eindeutig durch andere Eigenschaften bestimmen lässt (vgl. Abschnitt 5.1.3). Ein Beispiel für ein abgeleitetes Attribut stellt das Attribut `name` aus Listing 5.3 dar. Ein Beispiel für eine abgeleitete Assoziation ist die `produces`-Beziehung aus Listing 5.4. Zur Kennzeichnung von abgeleiteten Eigenschaften wird die `/`-Markierung verwendet. Zusätzlich wird eine Berechnungsvorschrift (vgl. Abschnitt 5.1.3) per OCL spezifiziert. Ob eine Eigenschaft abgeleitet werden kann oder deren Informationen direkt erhoben werden müssen, ist durch die Elemente des AMs vorgegeben. Ändert sich dort der Detailgrad, so können ggf. weitere Eigenschaften als abgeleitet markiert oder entsprechende Markierungen entfernt werden. Würde beispielsweise der `simpleName` aus Listing 5.3 entfernt werden, könnte `name` nicht länger berechnet werden. Stattdessen müsste die Markierung (`/`) entfernt und die zu `name` zugehörigen Daten direkt bei der Extraktion mit erhoben werden.

Da nicht nur Elemente, wie in Schritt (2) beschrieben, Spezialisierungen haben, sondern auch Assoziationen und Kompositionen speziellere Ausprägungen haben können, werden in Schritt (5) Assoziationsspezialisierungen spezifiziert. Durch die Möglichkeit verschiedene Abstraktionsstufen für Beziehungen und Elemente zu definieren und in einem Modell zu integrieren ist Anforderung *MRQ-3.2* erfüllt. Während es mit der Vererbungsbeziehung ein eigenes Modellierungskonzept zur Spezialisierung von Klassen eines CDs gibt, existiert ein solches Konzept zur Spezialisierung von Assoziationen nicht. Stattdessen kann eine solche Beziehung zwischen zwei Assoziationen durch OCL modelliert werden. Dazu wird spezifiziert, dass alle Elemente der spezialisierenden Assoziation in den Elementen der abstrakteren Assoziation enthalten sind. Die OCL bietet hierzu die `containsAll`-Operation für Mengen an [Rum16]. Ein Beispiel hierfür findet sich in Zeile 6 von Listing 6.2.

Unter der Annahme, dass Spezialisierungen nur für Assoziationen des AM-Kerns modelliert werden, ist eine alternative Modellierungstechnik denkbar. Dazu könnte der Kern als Metamodell für das projektspezifische AM dienen. Darüber hinaus könnten Multi-Level-Modeling-Techniken wie [dLG10] genutzt werden. Diese bieten die Möglichkeit, sowohl zwischen Elementen als auch zwischen Beziehungen eines Modells Instanziierungsbeziehungen zu definieren. Da durch die Verwendung solcher Techniken Anforderung *MRQ-1* nicht erfüllt würde, wurde von deren Verwendung an dieser Stelle abgesehen.

In Schritt (6) lassen sich per OCL Prüfbedingungen für das AM modellieren (vgl. Abschnitt 5.1.3), welche bestimmte Projektsituationen ausschließen. Artefaktbasierte Analysen können deshalb davon ausgehen, dass als Eingabe Artefaktdaten geliefert werden, die alle Prüfbedingungen erfüllen. Bei Verletzung einer solchen Bedingung muss zunächst die Ursache für die Verletzung behoben werden bevor artefaktbasierte Analysen durchgeführt werden können. Mögliche Ursachen für Inkonsistenzen zwischen Artefaktdaten und AM werden später in Abschnitt 8.4.1 betrachtet. Ein Beispiel für eine OCL-Prüfbedingung ist in Zeile 14 von Listing 6.3 dargestellt. Dort wird sichergestellt, dass Java-Quellcodedateien immer die Dateiendung "java" haben.

Als Sonderfall bei der Erstellung projektspezifischer AMe kann die Modellierung von Verhalten angesehen werden. Zur Spezifikation von Verhalten bietet die UML eigene Diagrammarten, wie Aktivitätsdiagramme (engl. *Activity Diagram* (AD)), Zustandsdiagramme (engl. *Activity*

Diagram (AD)) oder Sequenzdiagramme (engl. *Sequence Diagram* (SD)), an. In dieser Arbeit wird auf die Nutzung zusätzlicher Modellierungssprachen verzichtet. Abläufe werden innerhalb von ODs als eine Art Protokoll codiert (vgl. Abschnitt 6.5.1). Dadurch ist Anforderung *MRQ-4* direkt erfüllt, da durch die Verwendung von CDs zur Modellierung von Struktur und Verhalten durch OCL kombinierte Aussagen über Beides möglich sind. Die zugehörigen Artefaktdaten können alle im selben Format (OD) gespeichert werden, wodurch kein zusätzliches Transformieren der Daten notwendig ist. Anstatt eine weitere Modellierungssprache zur Beschreibung des Ablaufverhaltens während der Ausführung des Buildprozesses zu verwenden, wurde in Abschnitt 6.5 die Ausführung von Werkzeugen als Sequenz von hierarchisch strukturierten Aktionen modelliert. Ein Hinzufügen von weiteren Aktionen und Events ist mit der in diesem Abschnitt beschriebenen Methodik möglich.

7.2. Methodik zur Spezifikation projektspezifischer Analysen

Der zweite Schritt der Methodik sieht die Spezifikation von projektspezifischen Analysen basierend auf dem zuvor erstellten AM vor. Artefaktbasierte Analysen sollen nach Anforderung *TRQ-4.2* wiederholbar und automatisiert durchführbar sein. Aus diesem Grund ist eine Implementierung der Analysen notwendig. Diese Implementierung kann entweder direkt geschehen – in dem Fall hätte die durchführende Person beide Rollen *Analyst* und *Analysewerkzeugentwickler* – oder aber der *Analyst* spezifiziert die Analysen als Anforderungen für den *Analysewerkzeugentwickler*, der anschließend ein entsprechendes Analysewerkzeug implementiert.

In dieser Arbeit wird die Spezifikation von Analysen per OCL durchgeführt. Die Gründe hierfür sind:

- Durch die Verwendung der CD-Sprache bei der Spezifikation des AMs bietet sich eine Spezifikation von Analysen mittels OCL an, da sich die beiden Sprachen gut in Kombination verwendeten lassen.
- Die OCL wird bereits zur Definition von projektspezifischen AMs genutzt. Die Wiederverwendung bekannter Sprachen verringert den Einarbeitungsaufwand für *Analysten*, führt dazu, dass Analysespezifikationen auch für *Architekten* lesbar sind und ermöglicht Wiederverwendung bei der Werkzeugentwicklung.
- Die OCL hat eine mathematisch fundierte Semantik, wodurch sich Analysen präzise beschreiben lassen. Dies erfüllt Anforderung *MRQ-7*.
- OCL-Ausdrücke eignen sich als Eingabe für einen Generator, der diese automatisch in MDD-Werkzeuge umsetzen kann. Somit lässt sich dieser Implementierungsschritt automatisieren, wodurch sich der Aufwand für den *Analysewerkzeugentwickler* reduziert. Dies trägt zur Erfüllung von Anforderung *TRQ-1* bei.

Im Folgenden wird auf die Spezifikation von OCL-basierten Analysen im Detail eingegangen. Darüber hinaus werden Spezialformen solcher Analysen vorgestellt und abschließend gezeigt, wie sich die Nutzungsart eines OCL-Ausdrucks spezifizieren lässt.

7.2.1. OCL-basierte Analysen

Zur Spezifikationen von Analysen per OCL wird, wie bei der Spezifikation des AMs, zwischen zwei unterschiedlichen Arten von OCL-Ausdrücken unterschieden: Prüfbedingungen und Berechnungsvorschriften (vgl. Abschnitt 5.1.3). Für beide Ausdrucksarten können, wie in Abschnitt 6.6 gezeigt, Erweiterungen des CD-Anteils des verwendeten AMs notwendig sein. Eine Unterscheidung zwischen OCL-Ausdrücken des AMs und denen zu Analysezwecken ist bis hierhin nicht möglich, da sie sich syntaktisch nicht voneinander unterscheiden. Eine Möglichkeit zur Unterscheidung wird später durch die Einführung von Tags in Abschnitt 7.2.5 gegeben. Bei der Durchführung der Analysemethodik (vgl. Abbildung 7.2) muss aber zwischen den verschiedenen Arten unterschieden werden können, da (1) Analysen nur auf konsistenten Daten durchgeführt werden sollen und (2) auf die Ergebnisse der Prüfungen unterschiedlich reagiert werden soll. Bei Nichterfüllung von Prüfbedingungen des AMs soll die Durchführung abgebrochen werden, während bei der Prüfung zu Analysezwecken zwar ein entsprechendes Analyseergebnis berechnet, die Analyse aber fortgesetzt werden soll. Aus diesem Grund ist die Kennzeichnung der unterschiedlichen Nutzungsarten von OCL-Bedingungen (siehe Abschnitt 7.2.5) erforderlich.

Neben Prüfbedingungen können auch Berechnungsvorschriften zu Analysezwecken genutzt werden. Dazu muss eine Ergänzung des CD-Anteils des verwendeten AMs modelliert werden, in der mindestens eine abgeleitete Eigenschaft enthalten ist. Die OCL wird dann genutzt, um die Berechnungsvorschrift der abgeleiteten Eigenschaften formal zu spezifizieren. Auch hier muss die Nutzungsart des OCL-Ausdrucks angegeben werden, um sie von den Berechnungsvorschriften des AMs unterscheiden zu können. In Abschnitt 7.2.2 werden Spezialformen dieser Analyseart vorgestellt.

7.2.2. Abhängigkeitsanalyse und Change-Impact-Analyse

Die beiden Analyseformen *Abhängigkeitsanalyse* (vgl. Abschnitt 3.4.1) und *Change-Impact-Analyse* (vgl. Abschnitt 3.4.2) lassen sich als OCL-basierte Analysen unter Verwendung von Berechnungsvorschriften spezifizieren, wodurch sich die Anforderungen *ARQ-2* und *ARQ-3* erfüllen lassen. In diesem Abschnitt werden beide Analyseformen unter Verwendung von `refersTo`-Beziehungen spezifiziert. Listing 7.3 und Listing 7.4 zeigen die entsprechenden Spezifikationen.

Es wird pro Analyse je eine neue abgeleiteten Assoziation (`dependencyAnalysis` bzw. `changeImpactAnalysis`) hinzugefügt. Dabei enthält `dependencyAnalysis` alle transitiv, über die `refersTo`-Beziehung erreichbaren Artefakte. `changeImpactAnalysis`

```
association /dependencyAnalysis                                    CD
  [*] Artifact -> Artifact [*];                                    «AM4A»

context Artifact inv:
  dependencyAnalysis == refersTo**;
```

Listing 7.3: OCL-basierte Spezifikation einer Abhängikeitsanalyse.

```
association /changeImpactAnalysis                                  CD
  [*] Artifact -> Artifact [*];                                    «AM4A»

context Artifact a1, a2 inv:
  a1.changeImpactAnalysis == {a2 | a2.refersTo**.contains(a1)};
```

Listing 7.4: OCL-basierte Spezifikation einer Change-Impact-Analyse.

enthält hingegen alle Artefakte, die transitiv das aktuelle Artefakt über eine `refersTo`-Beziehung referenzieren.

Mit Hilfe der in Abschnitt 7.1 beschriebenen Technik zur Modellierung von Assoziationsspezialisierungen lassen sich neben der gezeigten Variante für `refersTo`-Beziehungen auch `produces`-Beziehungen verwenden oder andere Assoziationsarten definierten und anschließend zur Spezifikation der beiden Analysearten nutzen. Darüber hinaus lassen sich auch sehr spezifische Analysen für konkrete Artefaktarten spezifizieren. Ein Beispiel für eine solche konkretere Analyse wurde bereits in Abschnitt 6.6.3 für `FreeMarkerTemplates` vorgestellt.

7.2.3. Architekturanalysen

Durch die Anforderungen *ARQ-4* und *ARQ-5* wird gefordert, dass sich sowohl Ist- als auch Differenzarchitekturen für MDD-Projekte berechnen lassen. Für beide Architekturanalysen gilt dabei, dass Module aus beliebigen Artefakten bestehen dürfen. Um eine Differenzarchitektur (vgl. Abschnitt 3.5) berechnen zu können, werden sowohl die Soll-Architektur als auch die Ist-Architektur als Eingaben benötigt. Um wiederum die Ist-Architektur berechnen zu können, muss eine Zuordnung von Artefakten zu Architektureinheiten stattfinden.

`Module` sind Teil des AM-Kerns (siehe Abschnitt 5.2) und können sowohl zur Modellierung der Soll-Architektur, als auch für die Zuordnung von Artefakten (im Folgenden *Moduldefinition* genannt) genutzt werden. Für die Modellierung der Soll-Architektur werden die erwarteten `reliesOnModule`-Beziehungen zwischen den Modulen spezifiziert. Ein Beispiel hierfür ist in Listing 7.5 gegeben.

In dieser Arbeit beschränkt sich die Spezifikation der Ist-Architektur auf die Definition erwarteter Beziehungen zwischen Modulen. Dabei kann eine solche Erwartung pro Beziehungs-

OD
«SollArc»

```
1 objectdiagram PrescriptiveArchitecture {
2
3   moduleA:Module{};
4   moduleB:Module{};
5   moduleC:Module{};
6
7   link intendedReference moduleA -> moduleB;
8   link intendedReference moduleB -> moduleC;
9 }
```

Listing 7.5: Beispielhafte Definition einer Soll-Architektur durch Spezifikation von Modulen und zugehörigen, erwarteten Beziehungen.

art festgelegt werden. Es existieren aber auch Ansätze, mit denen man neben der Definition von erwarteten Beziehungen auch verschiedene Modularten (Schichten, Komponenten, Subsysteme, etc.) nutzen kann, die jeweils speziellen Konsistenzregeln folgen [PKB13]. Darüber hinaus können komplexere Bedingungen für Soll-Architekturen auch per DSL spezifiziert werden [GMR15]. Aufbauend auf dieser Arbeit könnte eine Kombination von ODs und OCL genutzt werden, um komplexere Bedingungen zu spezifizieren. Alternativ könnte die Definition der Soll-Architektur auch per DSL oder GUI ermöglicht werden, um sie für den *Analysten* komfortabler zu gestalten.

Sowohl die Ist-Architektur, die Soll-Architektur als auch die Moduldefinition werden in dem entwickelten Ansatz als OD-Artefakt gespeichert. Dadurch ist es möglich, sowohl die Ist-Architektur als auch die Soll-Architektur in einen einzigen Datensatz zusammenzuführen. Die im Beispiel dargestellte `reliesOnModule`-Beziehung stammt aus der in Abschnitt 6.6.1 vorgestellten Architekturanalyse. Eine beispielhafte Zuordnung von Artefakten zu einem Modul ist in Listing 7.6 dargestellt.

Dabei ist es erlaubt, neben einzelnen Artefakten den Modulen auch Ordner zuzuweisen. Bei zugeordneten Ordnern werden alle beinhaltenden Artefakte und rekursiv die Artefakte aller Unterordner dem Modul hinzugefügt. Diese Kurzschreibweise ist somit ein weiterer Schritt zur Erfüllung von Anforderung *MRQ-8*. Die Berechnung der Ist-Architektur wurde bereits in Abschnitt 6.6.1 als eine der beispielhaften Analysen vorgestellt. Bei der dort vorgestellten Analyse ist die Zuweisung von Ordnern in der oben beschrieben Form nicht spezifiziert. Im Folgenden wird diese komfortablere Variante zur Berechnung der Ist-Architektur verwendet. Ausgehend von der spezifizierten Soll-Architektur und der berechneten Ist-Architektur kann wie in Abschnitt 6.6.2 beschrieben die Differenzarchitektur berechnet werden. Eine schematische Durchführung dieser Berechnung folgt in Abschnitt 7.6.

```
objectdiagram ModuleDefinition {                                    OD
                                                                    «MD»
  moduleA:Module{};

  dir:Directory {
    simpleName = "dir";
    fullName = "src/main/java/app/dir";
  };

  Main:JavaSourceFile {
    simpleName = "Main";
    fullName = "src/main/java/Main.java";
  };

  link artifact moduleA -> Main;
  link artifact moduleA -> dir;
}
```

Listing 7.6: Beispielhafte Zuordnung von Artefakten zu Modulen (Moduldefinition).

7.2.4. Metriken

Metriken bilden Eigenschaften auf einen Zahlenwert ab (vgl. Abschnitt 3.6). Die Spezifikation einer Metrik, d.h. der Berechnungsvorschrift für den repräsentativen Zahlenwert, lässt sich ebenfalls per OCL-Ausdruck spezifizieren, wodurch Anforderung *ARQ-6* erfüllt ist. Dazu ist eine Erweiterung des zu analysierenden AMs nötigt. Eine solche Erweiterung ist in schematischer Form in Listing 7.7 dargestellt.

```
  class ArtifactMetric {                                            CD
    /int numReferedBy;                                              «AM4A»
  }

  association target [*] ArtifactMetric -> Artifact [1];

  context ArtifactMetric inv:
    numReferedBy ==
      {Artifact a | a.refersTo.contains(target)}.size();
```

Listing 7.7: OCL-basierte Spezifikation einer Metrik.

In dem dargestellten Beispiel wird die Anzahl der Artefakte berechnet, die über die `refersTo`-Beziehung mit dem Zielartefakt der Metrikberechnung verknüpft sind. Dazu wird dem AM eine spezielle Metrik Klasse hinzugefügt (`ArtifactMetric`). Zusätzlich wird

zwischen dieser Klasse und der Klasse für die die Metrik berechnet werden soll, eine Assoziation hinzugefügt. Diese Assoziation muss mit `target` benannt werden. Dies ist notwendig, damit die Berechnung der Metriken für alle Elemente der über diese Assoziation verlinkten Elementart automatisch durchgeführt werden kann. Diese Maßnahme ist ebenfalls ein Schritt zur Erfüllung von Anforderung *MRQ-8*. Zusätzlich muss der zugehörige OCL-Ausdruck noch als Metrikberechnung gekennzeichnet werden (siehe Abschnitt 7.2.5). Alternativ könnte statt des OCL-Ausdrucks auch die entsprechende Assoziation speziell gekennzeichnet oder aber manuell modelliert werden. Mit der gewählten Variante müssen lediglich für OCL-Ausdrücke zusätzlich Informationen spezifiziert werden, was die Arbeit des *Analysten* erleichtert.

7.2.5. Nutzungsart von OCL-Ausdrücken spezifizieren

In den vorherigen Abschnitten wurden bereits verschiedene Arten der Nutzung von OCL-Ausdrücken vorgestellt. Um eine automatisierte Behandlung von OCL-Ausdrücken durch MDD-Werkzeuge zu ermöglichen, muss die Nutzungsart spezifiziert werden. Dies wird hier durch den Einsatz einer Taggingsprache ermöglicht. Taggingsprachen erlauben das Hinzufügen von Zusatzinformationen (*Tags*) zu Modellelementen existierender Modelle ohne dabei die Lesbarkeit der Modelle zu beeinträchtigen. Dazu werden die Tags in einem separaten Artefakt definiert. Somit trägt diese Form der Modellierung zur Erfüllung von Anforderung *MRQ-2* bei. In [GLRR15] wird eine Methodik zur Erstellung domänenspezifischer Taggingsprachen vorgestellt. Neben der eigentlichen Tagdefinition können dort auch Tagschemata definiert werden, wodurch die Menge der möglichen Tags und deren Anwendungsmöglichkeit per DSL komfortabel spezifiziert werden kann (Anforderung *MRQ-8*). Die möglichen Tagtypen, deren Attribute sowie die dadurch definierten Nutzungsarten lassen sich Tabelle 7.8 entnehmen.

OCL-Nutzungsart	**Tag (Attribut)**
Prüfbedingung	`check`
Berechnungsvorschrift	`derive`
Analyse per Prüfbedingung	`check` (`analysis`)
Analyse per Berechnungsvorschrift	`derive` (`analysis`)
Analyse per Metrik	`metric`

Tabelle 7.8.: Tags zur Angabe der Nutzungsart von OCL-Bedingungen.

Bei Verwendung der Tagtypen `check` und `derive` kann optional das Attribut `analysis` angegeben werden, wodurch zwischen Prüfbedingungen und Berechnungsvorschriften des AMs und denen zu Analysezwecken unterschieden werden kann. Zusätzlich kann für alle Tagtypen ein `manual`-Attribut angegeben werden. Dadurch lässt sich angeben, dass die entsprechenden OCL-Ausdrücke manuell durch den *Analysewerkzeugentwickler* implementiert werden soll. Ein Generator kann so markierte Bedingungen bei der Verarbeitung ignorieren, wodurch die Nutzungsart trotzdem, wie in den anderen Fällen auch, per Tag angegeben werden kann.

Damit OCL-Ausdrücke überhaupt getaggt werden können, müssen sie eindeutig referenzierbar sein. Die verwendete OCL-Sprache erlaubt es, Ausdrücke, wie in Listing 7.9 dargestellt, mit Namen zu versehen. Dieser Name (hier `deriveUnusedTools`) muss eindeutig sein, damit das Tag eindeutig zugeordnet werden kann.

```
1  context Analysis inv deriveUnusedTools:                    OCL
2    unusedTools == Tool.removeAll(
3      {te.tool | ToolExecution te in execs.action});
```

Listing 7.9: Definition der OCL-Bedingungen `deriveUnusedTool` zur Berechnung der ungenutzten Werkzeuge bei einer Werkzeugkettenausführung.

Eine beispielhafte Tagdefinition zur Angabe der Nutzungsarten von OCL-Ausdrücken ist in Listing 7.10 dargestellt. Dort wird zunächst die Konformität zu dem Tagschema `ArtifactBasedAnalysis` (siehe Anhang D) angegeben. Die Tagdefinition trägt den Namen `ArtifactModelTags` und taggt drei OCL-Bedingungen des Modells `ArtifactModel` unter Verwendung der unterschiedlichen Tagtypen.

```
1 conforms to ArtifactBasedAnalysis;                          TD
2
3 tags ArtifactModelTags for ArtifactModel {
4     tag javaName with check;
5     tag deriveUnusedTool with derive = [analysis];
6     tag numberOfRefferingArtifacts with metric = manual;
7 }
```

Listing 7.10: Beispielhaftes Tagdefinition zur Angabe der Nutzungsart von OCL Bedingungen.

Im Rest der Arbeit werden die verwendeten Tags nicht immer dargestellt. Aus dem Kontext der Darstellung und der zugehörigen Beschreibung geht in diesen Fällen hervor, welche Nutzungsart gemeint ist. Werden keine weiteren Angaben gemacht, so werden dargestellte OCL-Ausdrücke zur Definition von Prüfbedingungen und Berechnungsvorschriften des AMs, wie sie beispielsweise in den Abschnitten 6.1 bis 6.5 genutzt wurden, verwendet.

7.3. Datenextraktion

Die Datenextraktion bildet den ersten Teilschritt bei der Durchführung von Analysen. Die Artefakte, die in der aktuellen Projektsituation vorliegen, werden zusammen mit ihren Beziehungen erfasst. Um auf eine Beziehung schließen zu können, muss je nach Analyseverfahren auch der Inhalt entsprechender Artefakte betrachtet (statische Analyse) oder aber die bei der

Durchführung der zu analysierenden Werkzeugkette auftretenden Beziehungen beobachtet werden.

Welche Artefaktdaten und Beziehungen von Interesse sind, wurde vorher in dem zugehörigen AM spezifiziert. Die Artefaktdaten müssen so extrahiert werden, dass sie zu dem AM passen. Die Methodik sieht vor, dass nur Analysen auf zu dem AM passenden Daten durchgeführt werden. Die Artefaktdaten können dabei alle Daten enthalten, die sich statisch nach Durchlauf der Werkzeugkette analysieren lassen. Dies schließt auch die Artefakte mit ein, die bereits vor Durchlauf der Werkzeugkette im Projekt existieren. Der Zeitpunkt der Datenextraktion macht hier keinen Unterschied, da Eingabeartefakte bei Ablauf der Werkzeugkette nicht verändert werden (siehe Abschnitt 2.2). Darüber hinaus können die Daten alle Beziehungen enthalten, die sich bei einem Durchlauf der Werkzeugkette beobachten lassen.

Der `fullName` von Artefakten spielt später beim Mergen eine wichtige Rolle und muss auch bei der Verwendung mehrerer Extraktoren für gleiche Dateien gleich berechnet werden. Der `fullName` von Artefakten ist ein Dateipfad relativ zu einem Wurzelverzeichnis (vgl. Abschnitt 5.2). Bei der Extraktion muss deshalb auf eine einheitliche Verwendung des Wurzelverzeichnis geachtet werden.

Weiterhin ist es wichtig, dass jede extrahierte Artefaktdatendatei ein in sich konsistentes OD darstellt. Um dies zu erreichen und um ein besseres Verständnis der Daten zu ermöglichen, wird eine einheitliche Berechnung von OD-Objektidentifikatoren verwendet. Dadurch ist gewährleistet, dass (1) korrekte und einzigartige Identifikatoren innerhalb eines ODs vergeben werden, (2) Identifikatoren genutzt werden können, die möglichst gut den Inhalt des Elements wiedergeben und auch für Menschen lesbar sind und (3) ein einheitlicher Stil bei Erstellung der Identifikatoren verwendet wird. Diese Vorgaben verhindern jedoch nicht, dass Identifikatoren für verschiedene Elemente in verschiedenen Artefaktdatendateien doppelt vergeben werden. Gerade bei Verwendung von verschiedenen MDD-Werkzeugen zur automatischen Extraktion von Artefaktdaten lässt sich dies nur schwer vermeiden. Dieses Problem wird in dem späteren Mergeschritt (siehe Abschnitt 8.3) aufgelöst. Alternativ dazu könnten zumindest für Artefakte die global eindeutigen Pfade des Dateisystems verwendet werden. Dies würde jedoch zu sehr langen Identifikatoren führen, wodurch die Lesbarkeit der von Artefaktdaten reduziert würde, was wiederum Anforderung *MRQ-5* widerspricht.

In dieser Arbeit werden Objektidentifikatoren nach dem in Listing 7.11 dargestellten Schema berechnet. Bei diesem Schema handelt es sich lediglich um eine Berechnungsvorschrift und nicht um die zur Implementierung verwendete Grammatik. Die in Kapitel 8 vorgestellten Werkzeuge können valide ODs der OD-Sprache mit beliebigen Objektidentifikatoren verarbeiten. Das dargestellte Schema beschreibt nur das Format von durch Werkzeugen automatisch erstellten Objektidentifikatoren, bei denen es sich um eine Teilmenge aller in der OD-Sprache erlaubten Objektidentifikatoren handelt. Somit wird die von den entwickelten Werkzeugen akzeptierte Eingabesprache nicht weiter eingeschränkt, was insbesondere das Verarbeiten von beim Reporing erstellten ODs ermöglicht. Das in [HR17] verwendete Schema zur Berechnung von OD-Objektidentifikatoren für das Reporting stellt ebenfalls eine Teilmenge aller in der OD-Sprache erlaubten Objektidentifikatoren dar.

```
1  "@" simpleName ("!" nameExtension)? ("(!" int ")")?
```

Listing 7.11: Schema zur Berechnung von OD-Objektidentifikatoren.

```
1  @readme!txt
2  @readme!txt(!2)
3  @Temp
4  @Temp(!2)
```

Listing 7.12: Beispiel für Identifikatoren von Artefaktdaten.

Listing 7.12 zeigt vier Beispiele für Objektidentifikatoren von Artefaktdaten. Der Identifikator in Zeile 1 repräsentiert das Artefakt `readme.txt`. Der Identifikator in Zeile 2 repräsentiert ein weiteres `readme.txt`-Artefakt und lässt sich durch die zusätzliche Nummerierung von dem Identifikator in Zeile 1 unterscheiden. Ein Ordner mit dem Namen `Temp` erhält den Identifikator aus Zeile 3, während für einen weiterer Ordner mit demselben Namen der Identifikator aus Zeile 4 genutzt wird.

Zur Berechnung von Identifikatoren für Dateien werden somit deren Attribute `simpleName` und `nameExtension` (vgl. Abschnitt 5.2.1) genutzt. Bei Elementen ohne Namenserweiterungen (Paket, Ordner, etc.) wird der mittlere Teil der Berechnungsvorschrift ausgelassen. Dabei wird für Elemente, die kein `simpleName`-Attribut haben, ein anderes Attribut (z.B. `name`) entsprechend genutzt. Unzulässige Spezialzeichen müssen in allen Fällen zunächst ersetzt werden (analog zu [HR17]). Bei der Neuberechnung von Identifikatoren wird stets geprüft, ob ein berechneter Identifikator bereits existiert. Ist dies der Fall, wird eine Nummerierung beginnend mit zwei hinzugefügt.

Durch die Wahl der Berechnungsvorschrift ist es möglich, kurze Namen zu verwenden, durch die sich die Elemente gut identifizieren lassen. Für manuell erstellte OD-Dateien kann die vorgestellte Berechnungsvorschrift ebenfalls genutzt werden, ist jedoch optional. Wichtig ist in jedem Fall, dass auch dort auf die Eindeutigkeit der Identifikatoren geachtet wird.

7.4. Merge von Artefaktdaten

Das Mergen von Artefaktdaten ist dann notwendig, wenn die Daten auf mehrere Dateien verteilt vorliegen. Dies tritt z.B. bei Verwendung mehrerer kleiner MDD-Werkzeuge zur automatisierten Extraktion von Artefaktdaten auf. Beim Mergen werden die Artefaktdaten aus verschiedenen Quellen in ein gemeinsames OD zusammengeführt.

Bei der Extraktion kann es zu redundanten Daten kommen, denn die Artefaktdaten müssen nicht disjunkt auf die extrahierten ODs verteilt sein. Objekte, die das gleiche Element im zu untersuchenden Projekt darstellen, werden beim Mergen „verschmolzen“, sodass in der Ausgabe nur ein einzelnes Objekt pro extrahiertem Element vorkommt. Demgegenüber wird bei

Beziehungen zwischen Elementen der Artefaktdaten davon ausgegangen, dass diese disjunkt auf die Artefaktdatendateien aufgeteilt sind. Somit müssen Links beim Mergen nicht verschmolzen werden, sondern können durch Vereinigung zusammengeführt werden.

In Abschnitt 5.1 wurde die OD-Sprache vorgestellt. Eine wichtige Eigenschaft zugehöriger Modelle ist, dass die Identifikatoren von Objekten innerhalb einer Modelldatei eindeutig sind. Dies ist auch bei der, in Abschnitt 7.3 vorgestellte Berechnung von Objektidentifikatoren der Fall. Durch die Verteilung von Artefaktdaten auf mehrere Artefakte kann es passieren, dass (1) gleiche Elemente in unterschiedlichen Artefakten unterschiedliche Identifikatoren haben und, dass (2) verschiedene Elemente in verschiedenen Artefakten den gleichen Identifikator haben. Daraus folgt, dass Objektidentifikatoren nicht als Kriterium bei der Auswahl der zu verschmelzenden Elemente fungieren und, dass sie sich beim Mergen ändern können.

Die Änderung von Objektidentifikatoren beim Mergen kann dazu führen, dass OD-Links ebenfalls angepasst werden müssen. OD-Links, nutzen Objektidentifikatoren, um Quell- und Zielobjekte zu referenzieren. Bei einer Änderung eines Identifikators müssen diese Referenzen ebenfalls angepasst werden. Die Änderung eines Identifikators hat somit den Charakter eines Rename-Refactorings (siehe *Rename Method* [FB99]) bei dem sowohl die Definition eines benannten Elements (hier der Objektidentifikator) als auch die Referenzen darauf geändert werden.

Der Mergealgorithmus bekommt zwei flache ODs (siehe Abschnitt 2.4.1) als Eingabe und produziert ein flaches OD als Ausgabe. Zusätzlich wird ein Kriterium angegeben, nach dem Objekte zusammengeführt werden sollen. Der Algorithmus wird in den folgenden Schritten ausgeführt:

1. Zuordnung von zu verschmelzenden Objekten des gleichen Objekttyps aus den verschiedenen ODs anhand des bei Aufruf des Algorithmus angegebenen Kriteriums. Dabei kann pro Objekttyp spezifiziert werden, ob der Objektidentifikator oder ein einzelnes Attribut zur Zuordnung der Objekte genutzt werden.

2. Mergen der in Schritt (1) zugeordneten Objekte und Ausgabe in das Ziel-OD.

3. Kopieren von Objekten, die in Schritt (1) nicht zugeordnet wurden, in das Ziel-OD. Ggf. müssen dabei Objektidentifikatoren angepasst werden.

4. Kopieren aller Links in das Ziel-OD. Ggf. müssen dabei die Referenzen auf die Objektidentifikatoren angepasst werden.

Das Standardverhalten bei der Zuordnung gleicher Objekte in Schritt (1) ist die Zuordnung anhand des Objektidentifikators, d.h., zwei Objekte werden als gleich erkannt, wenn sie denselben Objektidentifikator haben. Eine weitere Möglichkeit für die Angabe des Kriteriums ist, zu jeder Klasse des zugehörigen AMs separat anzugeben, nach welchem Kriterium sie einander zugeordnet werden. Dadurch ist es möglich, dass zugeordnete Objekte einen unterschiedlichen Identifikator haben. Weiterhin muss auch die Anzahl der Attribute nicht zwingend übereinstimmen. Es kann sein, dass in verschiedenen Dateien jeweils eine unterschiedliche Anzahl an

Attributen desselben Objekts dargestellt ist. Wird für mehrere Klassen einer Klassenhierarchie des zugehörigen AMs das Standardverhalten überschrieben, so überschreibt die Konfiguration der Subklasse die Konfiguration der Oberklasse. Somit muss in diesem Fall die Klassenstruktur des AMs ebenfalls als Eingabe für den Algorithmus bereitgestellt werden.

Nach der oben beschriebenen Zuordnung gleicher Objekte werden diese in dem zweiten Schritt des Algorithmus verschmolzen. O_{merged} bezeichnet im Folgenden das Ergebnis der Verschmelzung zweier Objekte O_1 und O_2. Dabei gilt $O_1 \in OD_1$, $O_2 \in OD_2$ und $O_{merged} \in OD_{merged}$.

Zunächst wird geprüft, ob die Typen von O_1 und O_2 übereinstimmen und ob Attribute, die in beiden Objekten mit Werten belegt wurden, denselben Wert haben. Schlägt diese Prüfung fehl, bricht der Algorithmus unter Angabe einer Fehlermeldung ab. Ansonsten werden die Attribute und ihre Wertebelegung in O_{merged} vereinigt. Attribute, die sowohl in O_1 als auch O_2 vorkommen werden nur einmal zu O_{merged} hinzugefügt und der Typ von O_{merged} wird auf den Typ von O_1 und O_2 gesetzt. Der Identifikator von O_{merged} entspricht dem von O_1, also dem Objekt aus der ersten Datenquelle. Da diese Datenquelle ein korrektes OD sein muss, sind alle Identifikatoren aus der Datenquelle bereits eindeutig. Unterscheiden sich die Identifikatoren von O_1 und O_2 wird dies als Änderung des Identifikators von O_2 interpretiert und diese Informationen für einen späteren Schritt zwischengespeichert.

Objekte, die nicht im ersten Schritt zugeordnet wurden, werden in Schritt (3) ebenfalls nach OD_{merged} übertragen. Dazu werden zunächst alle Objekte aus OD_1 übernommen. Da die Identifikatoren dieser Objekte nicht in Konflikt mit anderen Identifikatoren stehen können, ist eine entsprechende Prüfung hier nicht notwendig. Bei dem anschließenden Übertragen eines Objekts O_2 aus der zweiten Datenquelle muss hingegen geprüft werden, ob bereits ein Objekt O_1 mit demselben Identifikator existiert. Ist dies der Fall muss der Identifikator geändert werden, bevor O_2 nach OD_{merged} übertragen werden kann. Zur Berechnung des neuen Identifikators, wird der Algorithmus aus Abschnitt 7.3 genutzt, der bereits bei der Datenerhebung zum Einsatz kommt. Zusätzlich werden diese Änderungen an Identifikatoren ebenfalls zwischengespeichert.

Im letzten Schritt werden die Links von OD_1 und OD_2 nach OD_{merged} kopiert. Dabei müssen Links aus OD_2, dessen Start und/oder Zielobjekt in einem der vorherigen Schritte einen neuen Identifikator zugewiesen bekommen haben, angepasst werden. Dazu werden die Referenzen auf die Identifikatoren entsprechend der gespeicherten Umbenennungen im Zuge des Kopiervorgangs aktualisiert.

Abbildung 7.13 zeigt ein Beispiel, bei dem Artefaktdaten aus zwei verschiedenen Datenquellen zusammengefügt werden. Dabei wurde das `fullName`-Attribut als Kriterium für die Zusammenführung gewählt.

Das Objekt `@A1` aus Artefakt $\texttt{OD}_1$ wird bei Ausführung des Algorithmus dem Objekt `@A2` aus Artefakt $\texttt{OD}_2$ zugeordnet. Die anderen beiden Objekte wurden hingegen nicht zugeordnet, da die Werte ihrer `fullName`-Attribute einzigartig sind. Bei Verwendung des Standardverhaltens zur Zuordnung gleicher Objekte würde der Mergealgorithmus für das dargestellte Beispiel das Objekt `@A1` aus Artefakt $\texttt{OD}_1$ dem Objekt `@A1` aus Artefakt $\texttt{OD}_2$ zuordnen. Entsprechendes wäre dann auch für die beiden Objekte `@A2` der Fall.

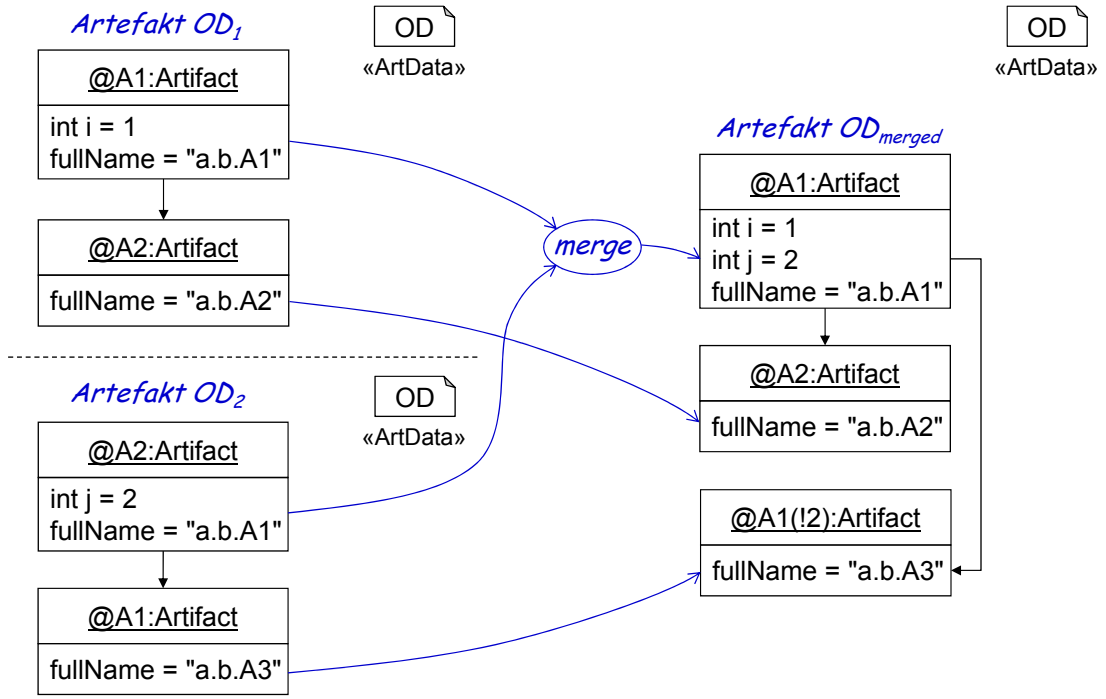

Abbildung 7.13.: Beispielhafte Durchführung des Mergealgorithmus unter Verwendung des `fullname`-Attributs als Zuordnungskriterium für Objekte.

Bei der Ausführung des Algorithmus wurde Objekt `@A1` aus Artefakt $\texttt{OD}_1$ mit Objekt `@A2` aus dem Artefakt $\texttt{OD}_2$ verschmolzen. Zusätzlich wurde das Objekt `@A1` aus $\texttt{OD}_2$ unter Verwendung Identifikatorschemas aus Abschnitt 7.3 in `@A1(!2)` umbenannt, weil bei dessen Übertragung nach $\texttt{OD}_{merged}$ bereits ein anderes Objekt mit dem Identifikator `@A1` im Ergebnis enthalten war.

Der hier vorgestellte Ansatz für das Zusammenfügen von Modellen ist speziell auf ODs zugeschnitten. Dort lässt sich zwar der erste Schritt, die Zuordnung zwischen Objekten, konfigurieren, die weiteren Schritte sind dann aber nicht mehr parametrierbar. Beispielsweise ist für eine alternative Vorgehensweise beim Zusammenfügen von Links auch eine Prüfung auf Duplikate denkbar.

Für das Zusammenführen von Modellen existieren auch allgemeinere Ansätze, welche verschiedene Mergeoperatoren unterstützen. In [BCE+06] wird ein Rahmenwerk vorgestellt, das den Mergeoperator als algebraischen Operator definiert. Mit Hilfe des Rahmenwerks werden dort Mergeoperationen für verschiedene Sprachen realisiert. Dabei werden jeweils Modelle der gleichen Sprache zusammengefügt. Ein Ansatz, mit dem sich ausgewählte Anteile aus verschiedenen Metamodellen zusammenfügen lassen, wird in [BHK+16] vorgestellt. Hier lassen sich modellspezifische Mergeoperationen unter Verwendung einer DSL angeben.

7.5. Datenprüfung und -anreicherung

Bei der Datenprüfung wird die Konformität der in den vorherigen Schritten extrahierten und zusammengefügten Daten zu dem zum Projekt zugehörigen AM geprüft. Bei einem positiven Ergebnis bei der Prüfung lassen sich die Daten anschließend durch die Berechnung abgeleiteter Eigenschaften anreichern und somit auf die eigentliche Analyse vorbereiten. Die Datenprüfung besteht dabei aus zwei Schritten: Die Prüfung der Konformität des Objektdiagramms zu dem CD-Anteil des AMs (*Strukturprüfung*) und die Überprüfung der OCL-Prüfbedingungen des AMs. Zusammen erfüllen die Teilprüfungen Anforderung *MRQ-9*. Für die Strukturprüfung lässt sich der in Abschnitt 5.1.4 beschrieben Algorithmus direkt nutzen. Dabei müssen je nach angenommener Semantik (Open World Assumption oder Closed World Assumption) die entsprechenden Schritte des Algorithmus ausgeführt werden. Im Folgenden wird zunächst auf die Prüfung der Konformität der Daten zu den OCL-Prüfbedingungen (Abschnitt 7.5.1) und anschließend auf die Anreicherung der Daten (Abschnitt 7.5.2) durch Auswertung der Berechnungsvorschriften eingegangen. Da die benötigten Berechnungen für beide Teilschritte mit Hilfe der OCL spezifiziert werden (vgl. Abschnitt 7.1), werden sie in diesem Abschnitt gemeinsam behandelt.

7.5.1. Auswertung von OCL-Prüfbedingungen

OCL-Prüfbedingungen liefern einen booleschen Wert zurück. Ein Beispiel für einen solchen OCL-Ausdruck aus dem in Kapitel 6 vorgestellten AM ist die Prüfung der Dateiendung von `JavaSourceFiles` (vgl. Listing 6.3). Listing 7.14 zeigt zugehörige Artefaktdaten die zu einer schematischen Prüfung genutzt werden.

In diesem Beispiel liefert die Auswertung der Prüfbedingung (Listing 6.3, Zeile 14) den Wert `false` zurück, da zwei der Java-Dateien mit "myJava" eine falsche Dateiendung haben. Zusätzlich zu diesem Wert kann im Fall einer Bedingungsverletzung auch ein Gegenbeispiel berechnet werden, d.h., es werden die Objektstrukturen ausgegeben, welche die Prüfbedingung verletzen. Ein solches Gegenbeispiel ist für den beschriebenen Fall in Listing 7.15 dargestellt.

In dem Gegenbeispiel sind genau die Artefaktdaten enthalten, die zur Verletzung der Bedingung geführt haben. Zusätzliche Informationen (hier die `reliesOnJava`-Links) werden hingegen nicht ausgegeben.

7.5.2. Anreicherung

Bei der Anreicherung werden abgeleitete Eigenschaften berechnet, die im AM per OCL-Berechnungsvorschrift spezifiziert wurden. Ein- und Ausgabe für diesen Teilschritt sind jeweils Artefaktdaten. Der Unterschied zwischen Ein- und Ausgabedaten besteht darin, dass bei den Ausgabedaten, die abgeleiteten Eigenschaften mit zugehörigen, berechneten Werten hinzugefügt wurden.

OD
«ArtData»

```
objectdiagram ExampleOCLInv {

  @A!myJava:JavaSourceFile{
    simpleName = "A";
    fullName = "p/A.myJava";
  };

  @B!java:JavaSourceFile{
    simpleName = "B";
    fullName = "p/B.java";
  };

  @C!myJava:JavaSourceFile{
    simpleName = "C";
    fullName = "p/C.myJava";
  };

  link reliesOnJava @A!myJava -> @B!java;
  link reliesOnJava @A!myJava -> @C!myJava;
}
```

Listing 7.14: Artefaktdaten als Eingabe zur schematischen Auswertung einer OCL-Prüfbedingung.

OD
«Report»

```
objectdiagram ExampleOCLInvResult {

  @A!myJava:JavaSourceFile{
    simpleName = "A";
    fullName = "p/A.myJava";
  };

  @C!myJava:JavaSourceFile{
    simpleName = "C";
    fullName = "p/C.myJava";
  };
}
```

Listing 7.15: Gegenbeispiel als zusätzliches Ergebnis bei negativer Auswertung einer OCL-Prüfbedingung.

In Listing 5.3 ist in Zeile 10 die Berechnung des abgeleiteten Attributs `name` dargestellt. Abbildung 7.16 zeigt eine beispielhafte Anreicherung von Artefaktdaten, bei der die Berechnung des Namens für die beiden dargestellten Objekte durchgeführt wird.

```
objectdiagram ExampleOCLInv {

  @A1:Artifact{
    simpleName = "A";
    nameExtension = "java";
  };

  @A2:Artifact{
    simpleName = "readme";
    nameExtension = "txt";
  };
}
```

OD «ArtData»

```
objectdiagram ExampleOCLInv {

  @A1:Artifact{
    simpleName = "A";
    nameExtension = "java";
    name = "A.java";
  };

  @A2:Artifact{
    simpleName = "readme";
    nameExtension = "txt";
    name = "readme.txt";
  };
}
```

OD «ArtData»

Abbildung 7.16.: Beispielhafte Anreicherung von Artefaktdaten.

7.6. Analyse von Artefaktdaten

Die Analyse der Artefaktdaten ist der letzte Teilschritt bei der Durchführung der artefaktbasierten Analyse. Hier werden die in Abschnitt 7.2 spezifizierten Analysen durchgeführt. Dazu werden angereicherte Artefaktdaten eingelesen und ausgewertet. Die Form des Ergebnisses hängt von der jeweiligen Analyseform ab. Bei der Durchführung einer Analyse per Prüfbedingung ist das Ergebnis ein boolescher Wert. Für die anderen in diesem Kapitel beschriebenen Analyseformen ist das Ergebnis hingegen wiederum ein OD. In den meisten Fällen beinhaltet dieses OD zu dem AM passende Artefaktdaten. Im Folgenden wird die Durchführung der verschiedenen Analysearten aus Abschnitt 7.2 beschrieben und schematisch durchgeführt. Zusätzlich wird die Datenfilterung als Analyseart eingeführt.

Analyse per Prüfbedingung Die Durchführung einer Analyse per Prüfbedingung geschieht analog zu der Auswertung von Prüfbedingungen zur Absicherung der Konsistenz zwischen Artefaktdaten und dem verwendeten AM (vgl. Abschnitt 7.5.1). Wird ein OCL-Ausdruck zur Analyse und nicht zur Konsistenzprüfung ausgewertet, kann aber eine andere Reaktion auf das Ergebnis der Auswertung erfolgen. Die in Kapitel 8 vorgestellte Werkzeugkette ermöglicht es, mehrere Analysen in einem Durchlauf durchzuführen, d.h., die Analyse wird auch dann fortgesetzt, wenn ein negatives Ergebnis bei der Auswertung einer Prüfbedingung zu Analysezwecken auftritt. Demgegenüber wird die Ausführung der Werkzeugkette bei Verletzung einer Prüfbedingung zum Zweck der Konsistenzsicherung abgebrochen.

Analyse per abgeleiteter Eigenschaft Die Analyse per Berechnung abgeleiteter Eigenschaften geschieht analog zum Vorgehen bei der Anreicherung. Um eine solche Analyse durchzuführen, muss zuvor die abgeleitete Eigenschaft, die das Analyseergebnis enthalten soll, als Erweiterung des AMs modelliert und deren Berechnung per OCL-Berechnungsvorschrift spezifiziert werden (vgl. Abschnitt 7.2.1). Diese Modellierung kann in einer separaten Datei geschehen, die sich nachträglich durch MDD-Werkzeuge mit dem AM zusammenführen lässt. Ein solches Vorgehen trägt zur Erfüllung von Anforderung *MRQ-2* bei, da somit Analysen separiert spezifiziert werden können und das AM nicht durch die Analysespezifikationen unnötig überfrachtet wird.

Nach der Vereinigung von Analysen und AM lässt sich das integrierte Ergebnis als Spezifikation für ein MDD-Werkzeug verstehen, welches zugehörige Artefaktdaten einliest und die abgeleitete Eigenschaft entsprechend der OCL-Berechnungsvorschrift berechnet.

Wie in Listing 7.4 gezeigt, lässt sich die Change-Impact-Analyse per OCL spezifizieren. Die dort spezifizierte Analyse wird für jedes Artefakt der übergebenen Artefaktdaten durchgeführt. Ein Beispiel für die Durchführung der Berechnung für ein einzelnes Artefakt ist in Abbildung 7.17 dargestellt.

In diesem Beispiel werden zur Berechnung der Change-Impact-Analyse für das Artefakt `@A5` alle eingehenden `refersTo`-Beziehungen betrachtet und dann eine rekursive Rückwärtssuche durchgeführt. Das Ergebnis für das gezeigte Beispiel sind die Artefakte `@A1`, `@A2` und `@A3`. Die zur Berechnung verwendeten `refersTo`-Links und die gefundenen Artefakte sind im oberen Teil der Abbildung hervorgehoben. Diese Artefakte werden durch berechnete, abgeleitete Links mit dem Ausgangsartefakt (`@A5`), wie unten dargestellt, verbunden. Dort sind allerdings nur neu hinzugefügte Links dargestellt. Tatsächlich werden die Eingabeartefaktdaten durch diese Links angereichert, d.h., die neu berechneten Links werden den Eingabedaten hinzugefügt. Darüber hinaus wird nach der Spezifikation aus Listing 7.4 die Analyse für jedes der Artefakte (`@A1` bis `@A5`) durchgeführt, was zu der Berechnung und dem Hinzufügen solcher Links für alle Artefakte bis auf Artefakt `@A1` führt. `@A1` hat keine eingehenden `refersTo`-Beziehungen, weshalb kein entsprechender Link berechnet wird.

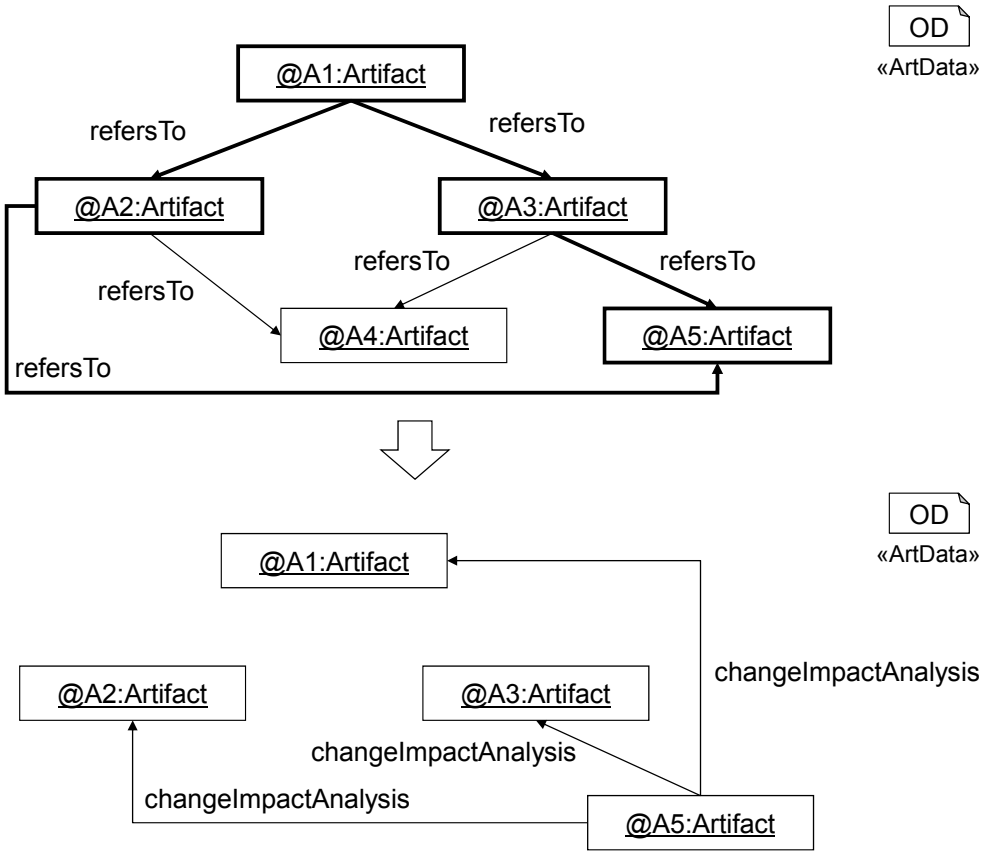

Abbildung 7.17.: Schematische Durchführung einer OCL-basierten Analyse am Beispiel der Change-Impact-Analyse für das Artefakt `@A5`.

.

Datenfilterung Bei der Filterung von Artefaktdaten können Elemente und Beziehungen anhand der zugehörigen Klassen oder Assoziationen (auch Kompositionen) des AMs gefiltert werden. Ein- und Ausgabe sind somit jeweils zu einem AM konsistente Artefaktdaten. Dabei kann es durch die Filterung dazu kommen, dass für die Ausgabe die Open World Assumption (vgl. Abschnitt 5.1.4) angenommen werden muss, auch wenn für die Eingabe die Close World Assumption (vgl. Abschnitt 5.1.4) angenommen werden konnte. Dies ist genau dann der Fall, wenn Objekte oder Links deren Existenz durch das AM gefordert werden, bei der Filterung entfernt wurden.

Der Algorithmus zur Filterung bekommt neben dem Eingabe-OD Informationen über die zulässigen Objekttypen und Linknamen als Parameter übergeben. Der Algorithmus ist wie folgt definiert:

1. Filterung der Objekte anhand der zulässigen Objekttypen und Ausgabe in das Ziel-OD.
2. Filterung der Links anhand der zulässigen Objekttypen und Linknamen und Ausgabe in das Ziel-OD.

Im ersten Schritt werden Objekte des Eingabe-ODs genau dann in das Ausgabe-OD kopiert, wenn deren zugehörige Objekttypen (Klasse des verwendeten AMs) als Parameter übergeben wurde. Analog dazu werden im zweiten Schritt Links genau dann in das Ausgabe-OD kopiert, wenn deren zugehörige Linknamen (Assoziationen des verwendeten AMs) als Parameter übergeben wurden. Zusätzlich muss im zweiten Schritt eine weitere Bedingung erfüllt sein, damit ein Link kopiert wird: Beide verlinkten Objekte (Start- und Zielobjekt des gerichteten Links) müssen im ersten Schritt bereits übertragen worden sein. Ist dies nicht der Fall, wird der Link auch dann nicht übertragen, wenn der zugehörige Linkname als Parameter vorliegt. Ohne diese Zusatzregel könnten Links in der Ausgabe hinzugefügt werden, deren Start- und/oder Zielobjekt bei der Filterung entfernt wurden. Ein solches OD wäre somit in sich inkonsistent, da es eine ungültige Referenz auf das entfernte Objekt enthält. Abbildung 7.18 zeigt eine beispielhafte Filterung von Artefaktdaten. Dabei wurden als Parameter für zulässige Objekttypen {`JavaSourceFile`} und für zulässige Linknamen {`defines`, `imports`} angegeben.

In diesem Beispiel wurde das Objekt `@A` entfernt, da es vom Type `Type` ist, welcher nicht als zulässiger Objekttyp definiert wurde. Analog dazu wurde der `reliesOnJava`-Link entfernt, da der zugehörige Linkname ebenfalls nicht als Parameter übergeben wurde. Der `defines` Link wurde hingegen entfernt obwohl `defines` in der Menge der zulässigen Linknamen enthalten ist, da das Zielobjekt `@A` In Schritt (1) entfernt wurde.

Die Datenfilterung wird im Gegensatz zu den anderen vorgestellten Analyseformen nicht per OCL spezifiziert. Stattdessen reicht es aus, eine Menge von Objekttypen und eine Menge von Linknamen als Parameter zu spezifizieren.

Differenzarchitektur Wie in Abschnitt 6.6.2 spezifiziert wurde, wird die Differenzarchitektur aus der Soll- und der Ist-Architektur gebildet. Die Soll-Architektur muss dazu, wie in Abschnitt 7.2.3 beschrieben, definiert werden. Die Ist-Architektur lässt sich aus einer Moduldefinition und zugehörigen Artefaktdaten berechnen. Abbildung 7.19 zeigt eine beispielhafte Spezifikation von drei Modulen, denen bereits Artefakte zugeordnet wurden. Zusätzlich wurden die bereits extrahierten Beziehungen zwischen den Artefakten dargestellt.

Aus diesen Informationen lässt sich nun die Ist-Architektur, wie in Abschnitt 6.6.1 beschrieben, berechnen. Das Ergebnis der Berechnung ist in Abbildung 7.20 unten links dargestellt.

Außerdem ist oben links in der Abbildung eine Soll-Architektur geben. Somit sind alle Informationen vorhanden, um die Differenzarchitektur zu bestimmen. Das Ergebnis ist auf der rechten Seite der Abbildung dargestellt. Es existieren zwei unerwartete, zusätzliche Beziehungen (`unintendedReference`) und darüber hinaus fehlt eine der erwarteten Beziehungen, was zu einem `missingReference`-Link in der Differenzarchitektur führt.

Metrikberechnung In Listing 7.7 wurde bereits eine Erweiterung des AM-Kerns und ein zugehöriger OCL-Ausdruck zur Berechnung der Metrik vorgestellt. Nutzt man nun die in Abbildung 7.17 dargestellten Artefaktdaten als Eingabe lässt sich für jedes der Artefakte ein

```
objectdiagram ExampleFilter {

  @A!java:JavaSourceFile{
    simpleName = "A";
    fullName = "p/A.java";
  };

  @A:Type{
    simpleName = "A";
    fullName = "p.A";
  };

  @B!java:JavaSourceFile{
    simpleName = "B";
    fullName = "p/B.java";
  };

  link defines @A!java -> @A;
  link imports @A!java -> @B!java;
  link reliesOnJava @A!java -> @B!java;
}
```

OD «ArtData»

```
objectdiagram ExampleFilter {

  @A!java:JavaSourceFile{
    simpleName = "A";
    fullName = "p/A.java";
  };

  @B!java:JavaSourceFile{
    simpleName = "B";
    fullName = "p/B.java";
  };

  link imports @A!java -> @B!java;
}
```

OD «ArtData»

Abbildung 7.18.: Beispielhafte Filterung von Artefaktdaten.

Metrik Objekt berechnen. Listing 7.21 zeigt das Ergebnis der Berechnung für drei der Artefakte (`@A1`, `@A3`, `@A5`).

Durch die Berechnung wird ein Metrikobjekt und ein `target`-Link erstellt. Zusätzlich wird das abgeleitete Attribut der hinzugefügten Metrikobjekte, wie durch den OCL-Ausdruck aus

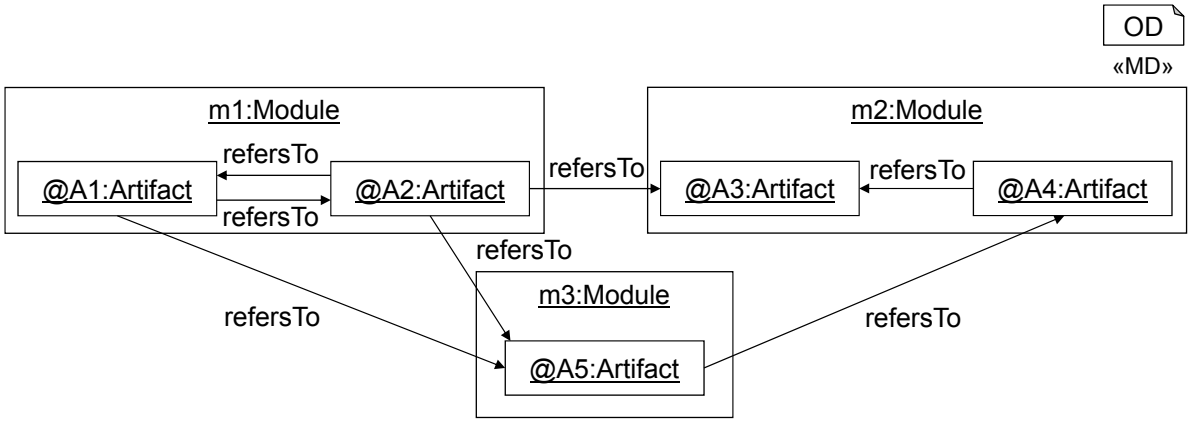

Abbildung 7.19.: Die Moduldefinition, zugeordnete Artefakte und deren Beziehungen, als Eingabe für die Berechnung der Ist-Architektur.

Listing 7.7 spezifiziert, berechnet und die zugehörigen, verlinkten Artefakte dem Ergebnis hinzugefügt.

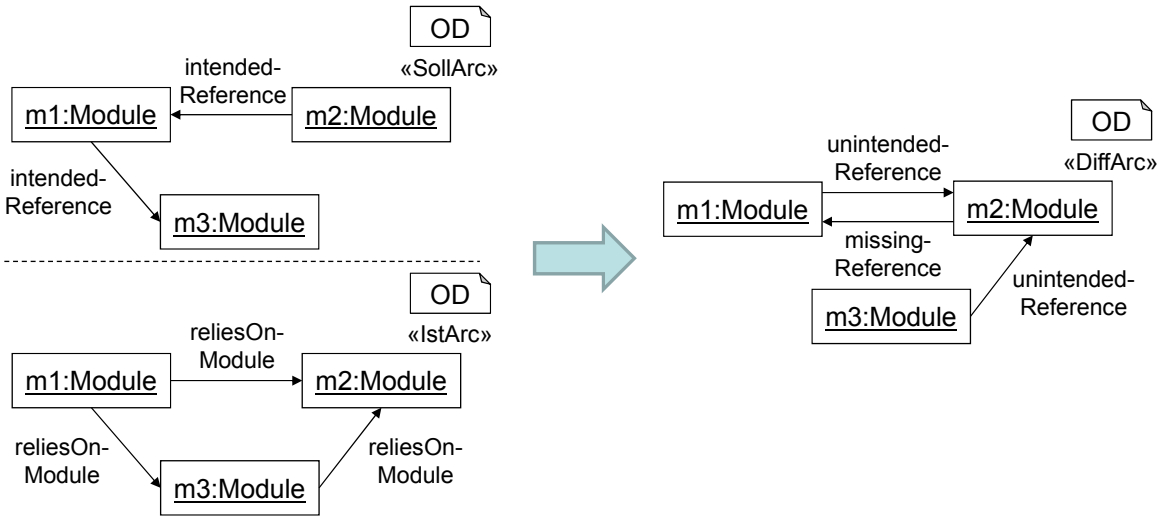

Abbildung 7.20.: Oben links: Soll-Architektur; Unten links: Berechnete Ist-Architektur; Rechts: Berechnete Differenzarchitektur aus Soll- und Ist-Architektur.

OD
«ArtData»

```
objectdiagram ExampleMetricResult {

  @A1:Artifact{};
  @A3:Artifact{};
  @A5:Artifact{};

  a1metric:ArtifactMetric {
    numReferedBy = 0;
  };

  a3metric:ArtifactMetric {
    numReferedBy = 1;
  };

  a5metric:ArtifactMetric {
    numReferedBy = 2;
  };

  link target a1metric -> @A1;
  link target a3metric -> @A3;
  link target a5metric -> @A5;
}
```

Listing 7.21: Ergebnis der beispielhaft durchgeführten Berechnung einer Metrik.

Kapitel 8.

MontiArT: Eine Werkzeugkette zur Durchführung artefaktbasierter Analysen

In Kapitel 7 wurde eine Methodik zur artefaktbasierten Analyse von MDD-Projekten vorgestellt. Um den Schritt der Durchführung von Analysen zu unterstützen, wurde als Teil dieser Arbeit die *MontiCore Artifact Toolchain* (MontiArT) entwickelt. MontiArT ist eine Werkzeugkette, die dazu genutzt werden kann Artefaktdaten zu erheben, zu mergen, zu validieren, anzureichern und schließlich zu analysieren. Somit werden alle Teilschritte der artefaktbasierten Analyse unterstützt.

Die einzelnen Schritte werden jeweils durch ein oder mehrere kleine Werkzeuge durchgeführt, die sich beispielsweise durch ein Skript zu einer Werkzeugkette zusammensetzen lassen. Aus diesem Grund trägt MontiArT auch den Begriff *Toolchain* im Namen. MontiArT wurde iterativ entwickelt wozu die Arbeiten [Hil14, Mai15, Ded16, Hil16, Käs16, Cuj17] beigetragen haben. Abbildung 8.1 zeigt eine beispielhafte Instanz der Werkzeugkette. Die Werkzeuge sind dabei anhand der durchzuführenden Schritte angeordnet.

Die Architektur als Werkzeugkette ist modular und anpassbar. Neue Werkzeuge lassen sich ohne Anpassungen anderer Werkzeuge einfach hinzufügen. Bestehende Werkzeuge lassen sich lokal anpassen oder können aus der Werkzeugkette entfernt werden ohne dass andere Werkzeuge angepasst werden müssen. Dies ist von Vorteil, da in dieser Arbeit davon ausgegangen wird, dass kein allgemeingültiges AM für alle MDD-Projekte existiert (siehe Abschnitt 4.4). Aus diesem Grund müssen bei Einsatz der Werkzeugkette in einem neuen Projekt üblicherweise projektspezifische Anpassungen vorgenommen werden. Durch die gewählte Architektur werden Anpassungen erleichtert und die Wiederverwendung einzelner Werkzeuge unterstützt. Dies trägt zur Erfüllung von Anforderung *TRQ-2* bei. Eine projektspezifische Instanz der Werkzeugkette wird im Folgenden MontiArT-Instanz genannt.

Definition 8.0.1 (MontiArT-Instanz)**.** *Mit dem Begriff MontiArT-Instanz wird eine projektspezifische, konfigurierte und zugeschnittene Instanz von MontiArT bezeichnet.*

Im Rest dieses Kapitels wird zunächst näher auf den verwendeten Werkzeugbegriff und die Eigenschaften von Werkzeugen eingegangen (siehe Abschnitt 8.1). Anschließend werden die

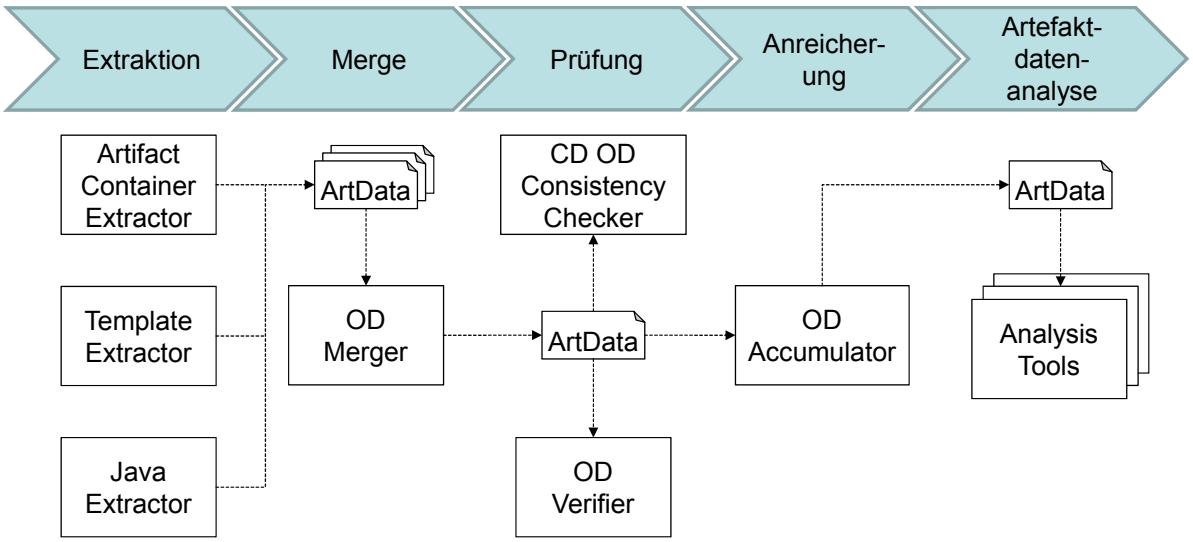

Abbildung 8.1.: Beispielhafte MontiArT-Instanz mit Zuordnung der Werkzeuge zu den Schritten der artefaktbasierten Analyse.

Werkzeuge zur Unterstützung der Analysedurchführung vorgestellt. Das Kapitel folgt dabei den in Kapitel 7 beschrieben und in Abbildung 8.1 dargestellten Teilschritten der artefaktbasierten Analyse, beginnend mit dem Extraktionsschritt. Abschließend werden noch Hilfswerkzeuge vorgestellt, die sich nicht eindeutig einem der Teilschritte zuordnen lassen. Eine Methodik zur Verwendung der Werkzeugkette wird später in Kapitel 9 beschrieben.

8.1. Werkzeuge

Bevor in den folgenden Abschnitten auf die MontiArT-Werkzeuge zur Unterstützung einzelner Schritte der Methodik zur Durchführung artefaktbasierter Analysen eingegangen wird, wird zunächst der Werkzeugbegriff beleuchtet und gemeinsame Eigenschaften der MontiArT-Werkzeuge vorgestellt.

Definition 8.1.1 (MontiArT-Werkzeug)**.** *Ein MontiArT-Werkzeug ist ein Werkzeug, dass einen für die artefaktbasierte Analyse notwendigen Arbeitsschritt durchführt.*

Bei der oben genannten Definition wird die Größe eines Arbeitsschritts nicht weiter eingeschränkt, d.h., es können sowohl ein Schritt, ein Teilschritt oder sogar mehrere Schritte des in Kapitel 7 dargestellten Ablaufs von einem Werkzeug durchgeführt werden. Im Folgenden wird der Begriff Werkzeug als Synonym für MontiArT-Werkzeug verwendet.

Allgemeine Eigenschaften von Werkzeugen wurden bereits in den Abschnitten 5.2.2 und Abschnitt 6.5.3 vorgestellt. Die Modellierung als AM hat zwar die Untersuchung der modellierten Elemente zum Ziel, trotzdem lassen sich die modellierten Eigenschaften von Werkzeugen hier auch auf die Analysewerkzeuge übertragen. So bestehen die Werkzeuge selbst aus Artefakten,

lesen Artefakte als Eingabe bei der Ausführung ein und produzieren Artefakte als Ausgabe der Ausführung.

Alle Werkzeuge werden als ausführbares Jar-Archiv zur Verfügung gestellt. Dabei handelt es sich um ein Uber-Jar [AMS17] oder auch Fat-Jar [FJE17]. Dies bedeutet, dass alle zur Ausführung benötigten Bibliotheken in dem Jar bereits enthalten sind. Zusätzlich enthalten die Jars eine `MANIFEST.MF`-Datei, welche die Klasse spezifiziert, die die zur Ausführung benötigte `main`-Methode enthält. Somit müssen bei dem Aufruf eines Werkzeugs weder zusätzliche Bibliotheken noch die Klasse mit der `main`-Methode angegeben werden, was die Nutzung der Werkzeuge vereinfacht. Damit ist Anforderung *TRQ-3.1* erfüllt, die eine komfortable, manuelle Nutzung der Werkzeuge fordert.

Des Weiteren können mehrere Werkzeuge in einem Artefakt gebündelt werden. Ein Beispiel dafür findet sich in den Abschnitten 8.4.4 und 8.4.5 wieder. Solche Fälle treten vor allem dann auf, wenn sich die entsprechenden Werkzeuge größere Teile des Quellcodes teilen und gemeinsam einen oder mehrere angrenzende Schritte des Ablaufs durchführen. Der Vorteil bei der Bündelung von Werkzeugen in ein gemeinsames Artefakt ist vor allem, dass die Gesamtgröße dieser Werkzeuge sinkt, da gemeinsam verwendete Bibliotheken nur einmal für mehrere Werkzeuge ausgeliefert werden müssen. Der Nachteil bei einer solchen Distribution ist, dass die enthaltenen Werkzeuge nicht auf unterschiedliche Versionen von Bibliotheken zurückgreifen können. Die Auswahl eines einzelnen Werkzeugs bei der Bündelung von Werkzeugen in einem gemeinsamen Jar-Archiv geschieht in diesen Fällen durch die Angabe des Werkzeugnamens als Parameter. Eine vollständige Liste aller mitgelieferten MontiArT-Werkzeuge und deren Verteilung auf die ausführbaren Jar-Archive befindet sich in Anhang E.

Die von MontiArT bereitgestellten Werkzeuge werden mit Hilfe des *Apache Maven Shade Plugins* [AMS17] zu ausführbaren Jar-Archiven gepackt. Die Jar-Archive folgen alle der folgenden Namenskonvention:

artifacts-<Kurzname>-<Version>-cli.jar

Im Rest der Arbeit wird, eine verkürzte Schreibweise für solche Archive verwendet. Dabei wird entweder nur der Kurzname oder aber <Kurzname>.jar verwendet.

8.1.1. Ein- und Ausgaben

Die Ein- und Ausgaben von Werkzeugen folgen ebenfalls bestimmten Konventionen. Die möglichen Ein- und Ausgaben der Werkzeuge sind in Tabelle 8.2 dargestellt. Bei der Vorstellung konkreter Werkzeuge in den späteren Abschnitten werden diese Angaben ergänzt.

Werkzeuge verarbeiten Artefakte und geben Artefakte als primäres Ergebnis aus. Somit lassen sich die in diesem Kapitel vorgestellten Werkzeug dateibasiert koppeln. Es gibt allerdings auch Werkzeuge, die kein Artefakt als primäre Ausgabe produzieren. Ein Beispiel hierfür ist ein Werkzeug, das die Konformität eines Modellartefakts zu einer Modellierungssprache prüft. Als Ergebnis wird dann das Resultat der Prüfung, nicht aber ein Artefakt zurückgegeben.

Neben dem Hauptergebnis können außerdem Berichte als Nebenprodukte erstellt werden. Diese enthalten Details über den Ablauf des Werkzeugs sind aber im Normalfall nicht für eine automatisierte Weiterverarbeitung vorgesehen. Im oben genannten Beispiel könnte bei einem negativen Prüfergebnis, die dafür Verantwortlichen Modellelemente in einem Report protokolliert werden. Darüber hinaus lassen sich Werkzeuge durch die zusätzliche Angabe von Optionen parametrisiert.

Neben der eigentlichen Ausgabe geben Werkzeuge ebenfalls einen Exitcode bei Beendigung der Ausführung zurück. Dieser orientiert sich an den verschiedenen Ergebnissen bei der Ausführung von JUnit [JUn18]. Dort zeigt ein *failure* an, dass die Ausführung des Tests zwar erfolgreich war, ein berechnetes Ergebnis aber nicht dem erwarteten Wert entspricht. Demgegenüber ist ein *error* ein Fehler der zum vorzeitigen Abbruch der Berechnung führt. Ist ein Test erfolgreich, so wird *success* zurückgegeben. Übertragen auf die Ausführung bei den Werkzeugen bedeutet dies, dass zwischen einem Abbruch der Ausführung, einer erfolgreichen Ausführung mit negativem Ergebnis und dem Erfolgsfall unterschieden wird.

Während ein *error* durch fehlerhaft formatierte Eingaben oder eine fehlerhafte Implementierung des Werkzeugs hervorgerufen werden kann, wird ein *failure* bei der Prüfung von OCL-Bedingungen mit negativem Ergebnis zurückgegeben. Die Exitcodes der Werkzeuge wurden in Analogie zum Verhalten der Java `main`-Methode gewählt, die bei erfolgreicher Terminierung eine 0 zurückgibt. Wird hingegen eine Exception geworfen, wird eine 1 zurückgegeben. Weiterhin werden in Java die textuellen Ausgaben von Fehlermeldungen, die durch das Auftreten einer Exception hervorgerufen werden, über die Standardfehlerausgabe ausgegeben. Wie in Tabelle 8.2 beschrieben orientiert sich MontiArT auch an dieser Stelle an der Java-Implementierung.

Durch Verwendung dieser Konventionen können Werkzeuge komfortabel von einem Benutzer aufgerufen (Anforderung *TRQ-3.1*) oder per Skript zu einer automatisch ausführbaren Werkzeugkette kombiniert werden (Anforderung *TRQ-3.2*). Eine Methodik zur Fehlererkennung innerhalb eines Skripts wird später in Abschnitt 9.3 vorgestellt.

	Name	Muss	Beschreibung
Eingaben	Artefakte	✓	Artefakte dienen als primäre Eingabe für Werkzeuge. Es wird mindestens ein Artefakt als Eingabe erwartet.
	Optionen		Zur Steuerung der Arbeitsschritte können Werkzeugen zusätzliche Optionen übergeben werden. Diese folgen der Konvention: Zwei vorangestellte Minuszeichen bei Verwendung von langen und ein vorangestelltes Minuszeichen bei Verwendung von kurzen Optionsnamen (z.B. `--help` und `-h`).
Ausgaben	Artefakte		Artefakte dienen als primäre Ausgabe der Werkzeuge. Trotzdem erzeugen nicht alle Werkzeuge Artefakte als Ausgabe.
	Exitcode	✓	Nach dem Durchlauf eines Werkzeugs wird der Status des Durchlaufs als Exitcode zurückgeben. Mögliche Status sind: • *success* (`System.exit(0)`): Erfolgreicher Arbeitsschritt; positives Ergebnis. • *error* (`System.exit(1)`): Vorzeitiger Abbruch des Arbeitsschritts. • *failure* (`System.exit(2)`): Erfolgreicher Arbeitsschritt; negatives Ergebnis. • *help* (`System.exit(3)`): Aufruf der Nutzungsinformationen (siehe Tabelle 8.4).
	Reports		Als weitere Ausgabe können Reportingartefakte entstehen. Diese dienen dazu, den Ablauf eines Arbeitsschrittes detailliert nachvollziehen zu können. Reports sind somit Nebenprodukte, die bei Ablauf eines Werkzeugs entstehen können.
	Textausgaben	✓	Informationen über den Status des Arbeitsschritts werden in textueller Form auf der Konsole ausgegeben. Da alle Werkzeuge sowohl in einem Skript zur automatisierten Analyse als auch einzeln von einem Benutzer verwendbar sein sollen, ist die Ausgabe von Informationen ebenfalls verbindlich. Informationen zu den Status *success* und *failure* werden über die Standardausgabe (*stdout*) ausgegeben, während Informationen zu dem Status *error* über die Standardfehlerausgabe (*stderr*) ausgegeben werden.

Tabelle 8.2.: Mögliche Ein- und Ausgaben von MontiArT-Werkzeugen.

8.1.2. Nutzung

Um die Werkzeuge auszuführen, müssen die entsprechenden ausführbaren Jar-Archive aufgerufen werden. Verbindliche Parameter werden dabei in der geforderten Reihenfolge direkt angeben. Im Normalfall werden so die Eingabe- und Ausgabeartefakte spezifiziert. Zusätzlich muss in dem Fall, dass mehrere Werkzeuge in einem Artefakt gebündelt sind, der Werkzeugname angegeben werden. Ein schematischer Werkzeugaufruf ist in Listing 8.3 dargestellt.

```
1 java -jar ExecutableJarArchive.jar ToolA Input.od Output.od
```

Listing 8.3: Aufruf von `ToolA` des ausführbaren Jar-Archivs `ExecutableJarArchive`.

Zusätzlich zu den notwendigen Parametern, können bei einem Werkzeugaufruf verschiedene Optionen angegeben werden. Solche Optionen sind werkzeugspezifisch, da sie den Ablauf eines speziellen Arbeitsschritts steuern. Trotzdem gibt es einige Standardoptionen, die in mehreren Werkzeugen Verwendung finden. Diese sind in Tabelle 8.4 dargestellt.

Kurz	Lang	Beschreibung
`-h`	`--help`	Zeigt den Hilfetext zur Verwendung des Werkzeugs an.
`-d`	`--reportdir`	Ändert das Verzeichnis in dem Reports abgelegt werden zu `reportDir`. Ist nur für Werkzeuge verfügbar, die einen Report ausgeben. Standardmäßig werden Reports in dasselbe Verzeichnis wie die Ausgabedatei(en) geschrieben.

Tabelle 8.4.: Standardoptionen von MontiArT Werkzeugen.

Optionen und die zugehörigen Werte werden als eigene Token bei Aufruf über die Kommandozeile angegeben. Bei der Vorstellung von werkzeugspezifischen Optionen im Rest dieses Kapitels wird eine ähnliche tabellarische Darstellungsweise verwendet. Per Hilfetext lassen sich die geforderten Parameter und verfügbaren Optionen auch direkt über einen Werkzeugaufruf abfragen. Dies trägt zur Erfüllung der in Punkt *TRQ-3.1* geforderten komfortablen Nutzbarkeit bei.

Um eine höhere Wiederverwendbarkeit auch in anderen Kontexten zu erreichen, wurden die Werkzeuge so konzipiert, dass sie sich ebenfalls programmatisch aufrufen lassen. Diese Nutzungsmöglichkeit wird in Abschnitt 8.1.3 vorgestellt.

8.1.3. Implementierung

In diesem Abschnitt wird auf die Klassenstruktur von ausführbaren Archiven und auf das verwendete Austauschformat zwischen Werkzeugen eingegangen. Die Klassenstruktur von ausführbaren Archiven ist in Abbildung 8.5 schematisch dargestellt.

`AToolCLI` stellt eine abstrakte Klasse dar, die zur Implementierung des *Command Line Interface* (CLI) des ausführbaren Jar-Archivs genutzt wird. Diese Klasse stellt Methoden bereit, mit denen ein bestimmter Zustand hergestellt werden kann. Basierend auf dem herbeigeführten

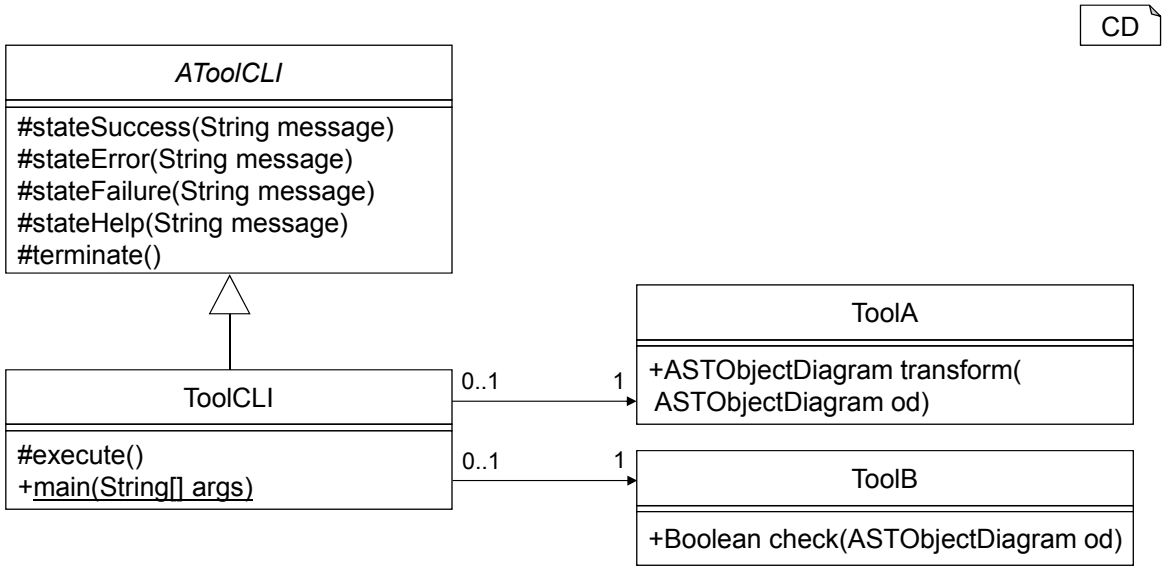

Abbildung 8.5.: Schematische Darstellung der Klassenstruktur für ausführbare Jar-Archive.

Zustand führt ein Aufruf der in Listing 8.6 dargestellten `terminate`-Methode zur Rückgabe eines entsprechenden Exitcodes (siehe Tabelle 8.2) sowie der Ausgabe des beim Herbeiführen des Zustands übergebenen Texts auf den entsprechenden Ausgaben (*strout* bzw. *strerr*).

Java

```
1  protected void terminate() {
2    if (stdMsg.isPresent()) {
3      System.out.println(stdMsg.get());
4    }
5
6    if (errorMsg.isPresent()) {
7      System.err.println(errorMsg.get());
8    }
9
10   System.exit(exitCode.getIntValue());
11 }
```

Listing 8.6: `terminate`-Methode der `AToolCLI`-Klasse.

Die Methoden der abstrakten Klasse `AToolCLI` können von der konkreten Klasse für das entsprechende ausführbare Jar-Archiv (hier `ToolCLI`) zur Implementierung einer Kommandozeilenschnittstelle (engl. *Command Line Interface* (CLI)) genutzt werden. Wie in Abbildung 8.5 dargestellt, wird auch, wenn mehrere Werkzeuge in einem einzelnen Archiv gebündelt werden, nur eine konkrete `ToolCLI`-Klasse benötigt. In dieser Klasse wird neben dem Setzen der Zustände auch die Parameterbehandlung sowie das Lesen und Schreiben von Dateien implementiert. Der Aufruf der einzelnen Werkzeuge wird in der `execute`-Methode dieser

Klasse implementiert. Diese Methode führt die Verarbeitung der Parameter durch und ruft anschließend die eigentliche Methode einer Werkzeugklasse (hier `ToolA` oder `ToolB`), welche die Bearbeitung des Arbeitsschritts ausführt, auf. Anders als die `execute`-Methode folgen Methoden der Werkzeugklassen keinem bestimmten Namensschema und können je nach Arbeitsschritt benannt werden. In Abbildung 8.5 sind dies die Methoden `transform` bzw. `check` der Klassen `ToolA` und `ToolB`. Beide Methoden bekommen ein OD in Form eines ASTs übergeben. Die `check`-Methode berechnet einen booleschen Wert als Ausgabe während die `transform`-Methode einen transformierten AST zurückgibt.

Durch die vorgestellte Aufgabenteilung können die konkreten Werkzeugklassen auch programmatisch genutzt werden, wodurch Anforderung *TRQ-3.3* erfüllt ist. Mehrere Werkzeuge lassen sich bei Bedarf als Unterprogramme zu einem großen Werkzeug zusammensetzen. Somit kann im Extremfall die ganze Werkzeugkette zur Analyse eines MDD-Projekts als ein einziges großes Werkzeug realisiert werden. Durch die flexiblen Möglichkeiten zur Nutzung und Kombination von Werkzeugen lassen sich diese auf die jeweiligen technischen Gegebenheiten bei der Analyse von MDD-Projekten anpassen.

Es ist zu empfehlen, dass auch bei Neuerstellung und Anpassung von projektspezifischen Analysewerkzeugen der *Analysewerkzeugentwickler* auf die Einhaltung der in diesem Abschnitt vorgestellten Konventionen und Richtlinien achtet. Die Einhaltung von Konventionen und die einheitliche Verwendung von Parametern und Interfacedefinitionen führt zu einer erhöhten Wartbarkeit von MontiArT und den projektspezifischen Instanzen. Dies führt zusammen mit der Wiederverwendung der bereits in der Klasse `AToolCLI` implementierten Funktionalität zu einem geringeren Implementierungsaufwand und trägt somit zur Erfüllung von Anforderung *TRQ-1* bei.

Als OD gespeicherte Artefaktdaten bilden das primäre Austauschformat von MontiArT-Werkzeugen, wodurch eine dateibasierte Kopplung [BJL+15] der Werkzeuge stattfindet. In der in dieser Arbeit beschriebenen Version von MontiArT (1.0.0), ist die verwendete Version der OD-Sprache nicht in allen Werkzeugen einheitlich. Einige Werkzeuge nutzen die neuere Version der Sprache (2.2.0), welche hierarchisch verschachtelte Objekte (siehe Abschnitt 2.4.1) unterstützt, während andere Werkzeuge noch mit der älteren Version der Sprache (0.2.2) arbeiten. Es kann pro ausgeliefertem Jar-Archiv nur eine Variante der Sprache genutzt werden, weshalb alle Werkzeuge, die in ein Jar-Archiv zusammengeführt wurden, auch die gleiche Version der Sprache nutzen müssen. Durch Transformationen lässt sich jedoch eine Abwärtskompatibilität herstellen, weshalb Werkzeuge, die mit verschiedenen OD-Versionen arbeiten, trotzdem zusammen genutzt werden können. Um die Abwärtskompatibilität zu erreichen, müssen aus einer OD-Datei der aktuellen Sprachversion die hierarchischen Anteile entfernt werden (vgl. Abschnitt 8.6.1). Weiterhin müssen neu eingeführte syntaktische Elemente, wie in diesem Fall Semikolons, nach Objektdefinitionen entfernt werden. Dies ist beispielsweise mit dem in Listing 8.7 dargestellten Shell-Skript möglich.

Bei Verwendung des Skripts muss der Pfad zur Eingabedatei in der Variable `$NewVersionODFile`, der Pfad zur Ausgabedatei in der Variable `$OldVersionODFile` enthalten sein. Der SED-Aufruf [TOG18] entfernt nun die entsprechenden Semikolons und schreibt

```SH
cat $NewVersionODFile | sed 's/}\s*;/}/g' > $OldVersionODFile
```

Listing 8.7: Shell-Skript zur Entfernung von Semikolons nach schließenden geschweiften Klammern bei Objektdefinition.

das Ergebnis in die Ausgabedatei. Bei Verwendung des in Listing 8.7 dargestellten Skripts, dürfen zwischen der geschweiften Klammer und dem Semikolon zwar Leerzeichen stehen, die Klammer und das Semikolon müssen sich jedoch in derselben Zeile befinden. Dies ist in MontiArT durch Verwendung des mitgelieferten PrettyPrinters der OD-Sprache erfüllt.

An dieser Stelle wird der Vorteil einer Architektur als Werkzeugkette sichtbar. Unterschiedliche Versionen der Sprache lassen sich in verschieden Werkzeugen nutzen. Auch wenn die Verwendung der veralteten Sprachversion langfristig entfernt werden sollte, lässt sich so eine Schrittweise Migration durchführen, sodass die Lauffähigkeit der gesamten Werkzeugkette zu keinem Zeitpunkt beeinträchtigt ist.

8.2. Datenextraktion

In diesem Abschnitt werden MontiArT-Werkzeuge zur Unterstützung der Datenextraktion, des ersten Teilschritts der in Kapitel 7 beschrieben Methodik zur Durchführung artefaktbasierter Analysen von MDD-Projekten, vorgestellt. Zu diesem Zweck enthält MontiArT einige Extraktoren, die in verschiedenen MDD-Projekten verwendet werden können.

Grundsätzlich lassen sich MontiArT-Extraktoren in zwei Gruppen einteilen: *Statische* und *dynamische* Extraktoren. Die Implementierungen der beiden Extraktorarten unterscheiden sich grundlegend voneinander.

So folgen die statischen Extraktoren strikt den in Abschnitt 8.1 vorgestellten Eigenschaften von MontiArT-Werkzeugen, während dies bei dynamischen Extraktoren nicht der Fall ist. Für statische Extraktoren wird eine Klassenstruktur vorgegeben, welche sich auch zur Implementierung für weitere projektspezifische Extraktoren verwenden lässt (siehe Abschnitt 8.2.1).

Die Implementierung von dynamischen Extraktoren hängt hingegen stärker von dem zu analysierenden Projekt ab. Hier müssen nicht nur die für die Analysen relevanten Konzepte durch ein AM modelliert werden, sondern auch der interne Aufbau des Extraktors muss an die zu analysierenden Werkzeuge angepasst werden, da interne Aktionen bei Ablauf der Werkzeuge mitprotokolliert werden müssen. Stehen diese Informationen nicht bereits in geeigneter Form zur Verfügung, kann auch eine Instrumentierung der Quellen notwendig sein (siehe Abschnitt 3.3.2). Somit sind dynamische Extraktoren zwar Werkzeuge nach der in Definition 8.1.1 gegeben Definition, erfüllen aber nicht die vorgestellten weiteren Eigenschaften der anderen Werkzeuge, da sie weder zwingend als ausführbares Jar-Archiv bereitgestellt werden, noch Artefakte als Eingaben bekommen müssen. Zusammenfassend gilt, dass für die Implementierung von dynamischen Extraktoren wenige allgemeingültige Richtlinien aufgestellt werden können,

da sehr flexibel auf die Gegebenheiten des zu untersuchenden Projekts eingegangen werden muss.

Trotzdem können die in Abschnitt 6.5 vorgestellten Konzepte als Vorlage für das Ausgabeformat der zu extrahierenden Artefaktdaten bei der Implementierung eines dynamischen Extraktors dienen. Ein Beispiel für einen dynamischen Extraktor wird in Abschnitt 10.1.2 vorgestellt. Dieser projektspezifische Extraktor ist speziell auf die Analyse des DEx-Generators zugeschnitten, weshalb er nicht als wiederverwendbarer Teil von MontiArT ausgeliefert wird. Für die Implementierung dynamischer Extraktoren zur Analyse von MontiCore Werkzeugen kann eine eigene Methodik angegeben werden. Diese basiert auf der Nutzung der Reportinginfrastruktur (siehe Abschnitt 2.3.4) und wird später in Abschnitt 9.2.3 vorgestellt. Somit werden sowohl dynamische als auch statische Extraktionsverfahren unterstützt, wodurch Anforderung *ARQ-1* erfüllt ist.

Im Folgenden wird zunächst in Abschnitt 8.2.1 der Aufbau statischer Extraktoren erläutert, bevor in den Abschnitten 8.2.2 bis 8.2.4 die mitgelieferten statischen Extraktoren vorgestellt werden.

8.2.1. Aufbau statischer Extraktoren

Die mitgelieferten statischen Extraktoren von MontiArT arbeiten allesamt dateibasiert. Dies bedeutet, dass sie Werkzeuge mit den in Abschnitt 8.1 vorgestellten Eigenschaften sind, Dateien als Eingabe einlesen und die extrahierten Artefaktdaten im OD-Format als Datei ausgeben. Die Festlegung der Eingabeartefakte erfolgt durch die Angabe eines Eingabepfades. Alle Artefakte, die in dem Pfad oder in einem im Pfad angegebenen Verzeichnis liegen, werden als Eingabeartefakt betrachtet. Teilpfade lassen sich durch ein Semikolon getrennt angeben.

Da die von MontiArT mitgelieferten statischen Extraktoren jeweils in einem eigenen Jar-Archiv ausgeliefert werden, ist bei deren Aufruf die zusätzliche Angabe eines Werkzeugnamens nicht notwendig. Um einen Extraktor von der Kommandozeile auszuführen, muss der Aufruf des ausführbaren Jar-Archivs dem in Listing 8.8 angegebenen Schema folgen.

```
1 java -jar <ExtractorName>.jar [OPTIONS] <inputPath> Output.od
```

Listing 8.8: Schematischer Aufruf eines statischen Extraktors.

Als Option lässt sich dabei, neben den Standardoptionen (vgl. Abschnitt 8.1.2) das Wurzelverzeichnis zur Berechnung des `fullName`-Attributs über die in Tabelle 8.9 dargestellte Option explizit angegeben.

Extraktorspezifische Parameter und Optionen werden in den folgenden Abschnitten bei der Beschreibung der konkreten Extraktoren separat angegeben. In der verwendeten MontiArT-Version nutzen alle Extraktoren bereits Version 2.2.0 der OD-Sprache.

Die Extraktoren folgen alle dem gleichen Aufbau und teilen sich eine Oberklasse, welche die gemeinsame Funktionalität bereitstellt. Abbildung 8.10 zeigt diesen Aufbau schematisch.

Kurz	Lang	Beschreibung
`-r`	`--rootdir`	Ermöglicht eine explizite Angabe des Wurzelverzeichnisse `rootDir`, welches zur Berechnung des `fullname`-Attributs extrahierter Artefakte genutzt wird.

Tabelle 8.9.: Mögliche Option beim Aufruf statischer Extraktoren.

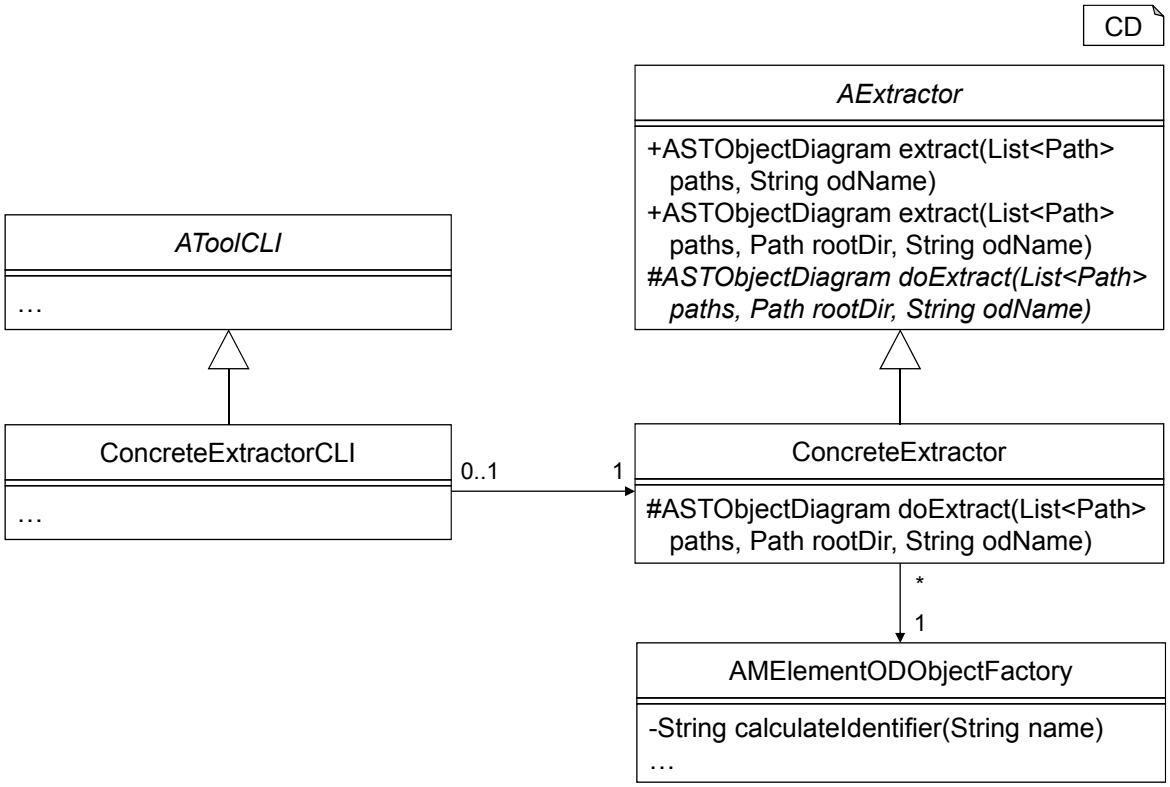

Abbildung 8.10.: Schematische Darstellung der Klassenstruktur der konkreten, statischen MontiArT-Extraktoren.

Die Methoden der beiden CLI-Klassen folgen dem vorgestellten Schema für Werkzeuge aus Abschnitt 8.1.3. Die statischen MontiArT-Extraktoren werden jeweils in einem eigenen ausführbaren Archiv ausgeliefert und haben somit jeweils eine eigene CLI-Klasse. Die zentrale Methode, die jeder konkrete Extraktor implementiert, ist `doExtract`. Innerhalb dieser Methode werden Dateien eingelesen, die Artefakttypen und deren Beziehungen zueinander extrahiert und abschließend als AST zurückgegeben. Jeder konkrete Extraktor extrahiert dabei einen Teil der im AM spezifizierten Informationen, d.h., die Implementierung der `doExtract`-Methode unterscheidet sich deutlich bei den einzelnen Extraktoren. Nichtsdestotrotz gibt es übergreifende Regeln, die bei den Implementierungen eingehalten werden müssen.

Zum Starten der Extraktion sollte immer eine der `extract`-Methoden der Klasse `AExtractor` aufgerufen werden. Die obere, der in Abbildung 8.10 dargestellten `extract`-

Methoden, berechnet das Wurzelverzeichnis aus den angegebenen Eingabepfaden. Das berechnete Wurzelverzeichnis entspricht dabei dem gemeinsamen Oberverzeichnis aller im Pfad angegebenen Verzeichnisse, dass (1) mit keinem der im Pfad angegeben Verzeichnisse übereinstimmt und (2) dabei die maximale Tiefe im Dateisystem hat. Wenn das so berechnete Wurzelverzeichnis durch eine ungünstige Pfadauswahl zu inkompatiblen Werten für den `fullName` bei Verwendung mehrerer Extraktoren führen würde, kann stattdessen die zweite `extract`-Methode aufgerufen werden. Dieser Methode lässt sich das zu verwendende (`rootdir`) als Parameter übergeben. Anschließend wir geprüft, ob das `rootdir` ein Oberverzeichnis aller im Pfad angegebener Teilpfade ist.

Beide `extract`-Methoden delegieren nach ihren jeweiligen Berechnungen die weitere Extraktion an die extraktorspezifische `doExtract`-Methode. Dieser Methode werden drei Parameter übergeben. Mittels `paths` wird die Liste an Eingabepfaden des Dateisystems übergeben. Nur die sich in den angegebenen Pfaden befindlichen Artefakte werden bei der Extraktion betrachtet. Der Parameter `odName` spezifiziert den Namen des berechneten ODs, während durch den Parameter `rootDir` das Wurzelverzeichnis zur Berechnung des `fullName`-Attributs (siehe Abschnitt 7.3) explizit angegeben werden kann.

Zur Berechnung des Objektidentifikators wird die Klasse `AMElementODObjectFactory` verwendet. Die Klasse stellt eine *Factory* im Sinne von [GHJV95] dar. Die Methode `calculateIdentifier` bekommt einen Namen übergeben und berechnet darauf basierend einen Identifikator nach der in Abschnitt 7.3 vorgestellten Berechnungsvorschrift. Die Methoden zur Erstellung von Objekten im OD-Format sind aus Platzgründen nicht in Abbildung 8.10 dargestellt.

8.2.2. Artifact Container Extractor

Der `ArtifactContainerExtractor` extrahiert zu den in Tabelle 8.11 dargestellten Anteilen des AMs zugehörige Artefaktdaten.

Klasse	Attribut	Assoziation
Artifact	simpleName fullName modified isRoot	
ArtifactContainer		contains

Tabelle 8.11.: Durch den `ArtifactContainerExtractor` extrahierte AM-Anteile.

Für alle Subklassen von `Artifact` werden dabei die angegebenen Attribute und für alle Subklassen von `ArtifactContainer` Links zu der angegebenen Assoziation extrahiert. Bei dieser Darstellung von Assoziationen gibt die Klasse (hier `ArtifactContainer`) jeweils den Ursprung der Assoziation an. Die Darstellungsform wird auch zur Beschreibung der extrahierten AM-Anteile noch folgender Extraktoren verwendet.

Neben den nichtabgeleitete Eigenschaften der `Artifact` Klasse wird somit das `fullName`-Attribute als einzige abgeleitete Eigenschaft bei der Extraktion mit berechnet. Dieses wird im Folgeschritt Merge von Artefaktdaten benötigt. Wie in Abschnitt 8.2.1 beschrieben, wird als `fullName` der relative Pfad ab dem Wurzelverzeichnis verwendet.

Zusätzlich zu den in Listing 8.8 dargestellten Parametern muss ein leeres Verzeichnis für temporäre Dateien angegeben werden, wodurch sich der Aufruf, wie in Listing 8.12 dargestellt, ändert.

```
1 java -jar ArtifactContainerExtraktor.jar [OPTIONS] <inputPath> \
2 Output.od <tempDir>
```

Listing 8.12: Schematischer Aufruf des `ArtifactContainerExtractors`.

Über die in Tabelle 8.9 dargestellten Optionen hinaus sind zwei weitere Optionen verfügbar. Diese werden in Tabelle 8.13 aufgelistet und erläutert.

Kurz	Lang	Beschreibung
`-a`	`--all`	Extrahiert alle Dateien, auch die, die nicht als Artefakte im AM aus Kapitel 6 spezifiziert sind. Die Dateiendung wird in diesem Fall als Artefakttyp der Objekte im Ausgabeartefakt verwendet.
`-d`	`--depth`	Extrahiert den Inhalt von Archiven rekursiv bis zu einer Tiefe von `depth`. Wird kein Wert für `depth` angegeben, so wird der Inhalt aller Archive rekursiv extrahiert.

Tabelle 8.13.: Mögliche Optionen bei Aufruf des `ArtifactContainerExtractors`.

Der `ArtifactContainerExtractor` extrahiert Dateien, Ordner und Archive, wobei Archive rekursiv durchsucht werden können. Der Extraktor führt dabei folgende Schritte durch:

1. Kopieren aller relevanter Artefakte aus dem Eingabepfad ab dem `rootDir` in das `tempDir`. Die relevanten Artefakte variieren je nach Wahl der Option `-a`.

2. Extrahieren aller Archive (rekursiv bis zur Tiefe `depth`).

3. Rekonstruieren des `modified`-Attributs (siehe Abschnitt 5.2) von Dateien und Ordner des Dateisystems.

4. Extrahieren der Artefaktdaten.

5. Ausgabe der Artefaktdaten in die Ausgabedatei.

Die Schritte (1)-(3) sind notwendig, um Archive rekursiv durchsuchen zu können. Schritt (4) lässt sich anschließend komfortabel unter Verwendung eines `FileVisitors` [Ora18] durchführen. Wie sich aus dem Namen der Klasse schließen lässt, handelt es sich dabei

um die Java-Implementierung des Visitorentwurfsmusters [GHJV95] zum Traversieren des Dateisystems.

Die Artefaktdaten werden schließlich in Form eines hierarchischen ODs (vgl. Abschnitt 2.4.1) ausgegeben. Die einzige extrahierte Beziehung ist die `contains`-Beziehung, die als hierarchische Verschachtelung realisiert wird.

8.2.3. Template Extractor

Der `TemplateExtractor` extrahiert zu den in Tabelle 8.14 dargestellten Anteilen des AMs zugehörige Artefaktdaten.

Klasse	Attribut	Assoziation
FreeMarkerTemplate	simpleName fullName	reliesOnJavaArtifact reliesOnTemplate
JavaSourceFile	simpleName fullName	

Tabelle 8.14.: Durch den `TemplateExtractor` extrahierte AM-Anteile.

Es werden nur die von Templates ausgehenden Beziehungen sowie die zugehörigen Artefakte extrahiert. Für die Artefakte wird dabei jeweils nur eine Teilmenge der Attribute, bestehend aus `simpleName` und `fullName`, extrahiert. Die anderen Attribute lassen sich durch ein Zusammenführen mit den extrahierten Daten des `ArtifactContainerExtractors` erhalten. Dazu wird bereits das `fullName`-Attribut benötigt. Der `simpleName` wird später zur Visualisierung (siehe Abschnitt 8.5.5) genutzt. Durch die Extraktion dieses Attributs lassen sich die extrahierten Daten auch ohne ein Zusammenfügen visualisieren. Aus den genannten Gründen werden auch bei den noch folgenden Extraktoren für Artefakte mindestens die Attribute `simpleName` und `fullName` extrahiert.

Der `TemplateExtractor` betrachtet nur FreeMarker-Templates als Eingabedateien. Der Aufruf des Extraktors folgt dem in Abschnitt 8.1.2 angegebenen Schema. Dabei sind nur die ebenfalls in Abschnitt 8.1.2 beschriebenen Standardoptionen für Werkzeuge sowie die Standardoption für statische Extraktoren (siehe Abschnitt 8.2.1) verfügbar. Da der `TemplateExtractor` einen Report ausgibt, kann zusätzlich das `reportDir` konfiguriert werden.

Abbildung 8.15 zeigt einen Templateausschnitt eines Beispieltemplates `TemplateA` sowie die daraus resultierenden Artefaktdaten als Ergebnis der Extraktion durch den `TemplateExtractor`.

Die in Zeile 1 des Templates aus Abbildung 8.15 dargestellte Anweisung führt zur Instanziierung einer Java Klasse. Entsprechend wird hier eine `reliesOnJavaArtifact`-Beziehung extrahiert. Die in Zeile 2 des Templates aus Abbildung 8.15 dargestellte Anweisung führt hingegen zum Aufruf eines weiteren Templates. Dementsprechend wird an dieser Stelle eine `reliesOnTemplate`-Beziehung extrahiert. Neben der dargestellten Variante, existieren in MontiCore weitere Varianten für Aufrufe zwischen Templates [HR17], deren Extraktion

FTL

```
<#assign helper = tc.instantiate("helpers.HelperA", ast)>
${tc.include("templates.TemplateB", ast)}
```

OD
«ArtData»

```
objectdiagram ExampleTemplateResult {

  @TemplateA!ftl:FreeMarkerTemplate{
    simpleName = "TemplateA";
    fullName = "templates/TemplateA.ftl";
  };

  @HelperA!java:JavaSourceFile{
    simpleName = "HelperA";
    fullName = "helpers/HelperA.java";
  };

  @TemplateB!ftl:FreeMarkerTemplate{
    simpleName = "TemplateB";
    fullName = "templates/TemplateB.ftl";
  };

  link reliesOnJavaArtifact @TemplateA!ftl -> @HelperA!java;
  link reliesOnTemplate @TemplateA!ftl -> @TemplateB!ftl;
}
```

Abbildung 8.15.: Beispielhafte Extraktion von Artefaktdaten mit dem `TemplateExtractor` aus einem Templateausschnitt.

ebenfalls unterstützt wird. Bevor eine der extrahierten Beziehungen der Ausgabe hinzugefügt wird, wird geprüft, ob ein entsprechendes Artefakt im Eingabepfad existiert. Ist dies nicht der Fall, wird das fehlende Artefakt zusammen mit der Beziehung in einem Report anstelle der Ausgabedaten hinzugefügt. Dieser Report wird ebenfalls im OD-Format erstellt und enthält nach dem Arbeitsschritt alle Beziehungen und Artefakte, die bei der Extraktion nicht gefunden werden konnten.

8.2.4. Java Extractor

Der `JavaExtractor` extrahiert zu den in Tabelle 8.16 dargestellten Anteilen des AMs zugehörige Artefaktdaten.

Der Aufruf des `JavaExtractors` folgt dem in Abschnitt 8.1.2 angegebenen Schema. Als mögliche Optionen lassen sich lediglich die Standardoptionen für Werkzeuge (siehe Abschnitt 8.1.2) und die Standardoption für statische Extraktoren (siehe Abschnitt 8.2.1) angeben. Da der `JavaExtractor` keinen Report erzeugt, ist die Option zur manuellen Angabe des `reportDirs` nicht verfügbar.

Klasse	Attribut	Assoziation
JavaSourceFile	simpleName fullName	belongsTo defines imports reliesOnJavaArtifact
Package	name fullName	contains subPackage
Type	name fullName	innerType reliesOn

Tabelle 8.16.: Durch den `JavaExtractor` extrahierte AM-Anteile.

Im Unterschied zu den anderen Extraktoren nutzt der `JavaExtractor` mit HUSACCT [PKvdWB14] eine externe Bibliothek zur Datenextraktion. HUSACCT ist eine Anwendung die sich in der verwendeten Version 5.2 nur per graphischer Benutzeroberfläche verwendet lässt, nicht aber programmatisch oder von der Kommandozeile aus genutzt werden kann. Durch geringe Modifikationen lässt sich dies jedoch ändern, sodass sich die *Analysekomponente* [HUS17] einzeln und programmatisch verwenden lässt. Mit Hilfe dieser Komponente lassen sich die in [PvdW15] beschriebenen Beziehungen zwischen Java Klassen extrahieren. Als Ergebnis wird ein Modell im XML-Format zurückgegeben, welches der `JavaExtractor` speicherintern zu AM konformen Artefaktdaten transformiert.

8.3. Merge von Artefaktdaten

In diesem Abschnitt wird die werkzeugtechnische Unterstützung des zweiten Schritts der Methodik zur Durchführung artefaktbasierter Analysen von MDD-Projekten beschrieben: Merge von Artefaktdaten. Zu diesem Zweck liefert MontiArT ein Werkzeug mit, mit dessen Hilfe sich dieser Schritt vollständig und automatisiert durchführen lässt. Somit ist im Gegensatz zur Datenextraktion keine Anpassung der Werkzeugkette durch das Hinzufügen von projektspezifischen Werkzeugen vorgesehen.

Im Folgenden wird zunächst das Werkzeug `ODMerger` vorgestellt. Da es sich bei diesem Werkzeug um ein in verschiedenen Kontexten wiederverwendbares Werkzeug handelt, wird dessen Nutzung im Rahmen von MontiArT anschließend in 8.3.2 beschrieben.

8.3.1. OD Merger

Der `ODMerger` ist ein Werkzeug zum Merge von ODs. Als Eingabe erwartet dieses Werkzeug mehrere OD-Dateien. Als Ausgabe wird eine OD-Datei mit den zusammengeführten Daten erstellt. Die Ein- und Ausgaben des `ODMergers` sind ODs der Sprachversion 0.2.2.

Der `ODMerger` wird zusammen mit anderen Werkzeugen in dem gemeinsamen, ausführbaren Archiv `language-tools` ausgeliefert. In diesem Archiv befinden sich Werkzeuge,

die nicht auf einem konkreten AM-basieren und sich deshalb für alle MontiArT-Instanzen nutzen lassen. Darüber hinaus ist auch eine Verwendung in Kontexten außerhalb der artefaktbasierten Analyse denkbar. Der `ODMerger` wird durch Angabe des Werkzeugnamens, wie in Listing 8.17 dargestellt, aufgerufen.

```
1 java -jar lang-tools.jar ODMerger [Options] Input1.od Input2.od \
2 Output.od
```

Listing 8.17: Schematischer Aufruf des `ODMergers`.

Zusätzlich zu den Standardoptionen für Werkzeuge (siehe Abschnitt 8.1.2) stehen für einen Aufruf des `ODMergers` die in Tabelle 8.18 dargestellten Optionen zur Verfügung.

Kurz	Lang	Beschreibung
`-f`	`--fullname`	Nutzt das `fullName`-Attribut als Kriterium für die Zuordnung von gleichen Elementen beim Mergen.
`-c`	`--config`	Das Kriterium für die Zuordnung gleicher Elemente pro Objekttyp lässt sich per `configFile` konfigurieren.

Tabelle 8.18.: Mögliche Optionen bei Aufruf des `ODMergers`.

Der `ODMerger` führt den in Abschnitt 7.4 beschriebenen Algorithmus zur Zusammenführung zweier ODs durch. Durch mehrfache Ausführung des Algorithmus lassen sich aber auch mehr als zwei ODs zusammenfügen. Dabei kann, anders als in Abschnitt 7.4 beschrieben, in der aktuellen Version (1.0.0) des Werkzeugs kein CD als Ergänzung der Konfiguration mitgegeben werden. Somit lassen sich aktuell nur konkrete Klassen des AMs ohne Betrachtung der Klassenhierarchie in der Konfiguration angeben. Die Übergabe des Klassendiagrammanteils des AMs ist als Erweiterung geplant.

Wie in Abschnitt 7.4 beschrieben, findet die Zuordnung von zusammenzuführenden Objekten standardmäßig anhand ihres Identifikators statt. Durch die Verwendung der in Tabelle 8.18 dargestellten Optionen lässt sich dieses Verhalten konfigurieren. Die Option `-f` bzw. `--fullname` ist eine praktische Alternative, wenn die Zuordnung aller Elemente ausschließlich über das `fullName`-Attribut stattfindet. Ist dies nicht der Fall kann durch die Angabe einer Konfigurationsdatei ein flexiblerer Ansatz gewählt werden. Diese Textdatei wird, wie in Listing 8.19 dargestellt, spezifiziert.

```
1 TextFile=simpleName                                          Conf
2 JavaSourceFile=fullName
```

Listing 8.19: Beispielhafte Spezifikation einer Konfigurationsdatei für den `ODMerger`.

In dem dargestellten Beispiel werden Objekte des Typs `TextFile` anhand des Attributs `simpleName` einander zugeordnet. Für Objekte des Typs `JavaSourceFile` wird hingegen

das `fullName`-Attribut genutzt. Für alle anderen Elemente wird die hier standardmäßige Zuordnung anhand des Objektidentifikators verwendet.

Zusätzlich zum Ausgabe-OD wird ein Report erstellt. Dieser listet die Objekte auf, bei denen sich der Identifikator geändert hat. Dabei wird neben dem ursprünglichen Identifikator auch der neue Identifikator angegeben.

8.3.2. Verwendung des OD Mergers in MontiArT

Der `ODMerger` ist ein in verschiedenen Kontexten wiederverwendbares Werkzeug. In diesem Abschnitt wird die Verwendung des `ODMergers` in MontiArT beschrieben.

Der `ODMerger` wird in MontiArT dazu verwendet, Artefaktdaten die als Ergebnis des Extraktionsschritts vorliegen, zusammenzuführen. Da der `ODMerger` mit der älteren Version der OD-Sprache (Version 0.2.2) arbeitet, müssen auch die Werkzeuge zur Unterstützung der nachfolgenden Schritte, diese Sprachversion nutzen. Durch die verteilte Extraktion der Artefaktdaten kann es, wie in Abschnitt 7.4 beschrieben, vorkommen, dass verschiedene Artefakte in den verschiedenen Datenquellen durch Objekte mit den gleichen Objektidentifikatoren oder gleiche Artefakte durch Objekte mit unterschiedlichen Identifikatoren repräsentiert werden. Aus diesem Grund ist ein einfaches Zusammenfügen per Objektidentifikator nicht möglich. Stattdessen werden Objekte durch ihr eindeutiges `fullName`-Attribut einander zugeordnet. Dies wird an dieser Stelle durch Auswahl der Option `-f` erreicht.

Damit das Zusammenführen inhaltlich korrekt funktioniert, muss als weitere Voraussetzung bei Nutzung des `ODMergers` im Schritt des Mergens sichergestellt werden, dass ein eindeutiger `fullName` bereits berechnet wurde. Zusätzlich muss sichergestellt sein, dass der Inhalt der anderen Attribute durch die verschiedenen Extraktoraufrufe für gleiche Elemente gleich berechnet wird. Dies wird, wie in Abschnitt 8.2.1 beschrieben, durch die Standard-MontiArT-Extraktoren gewährleistet. Zu beachten ist jedoch, dass bei Durchführung der Extraktoren ein einheitliches Wurzelverzeichnis genutzt werden muss.

Der zweite Anwendungsfall des `ODMergers` innerhalb von MontiArT ist dessen Verwendung als Vorbereitung für die Berechnung der Ist-Architektur aus einer Moduldefinition und zugehörigen Artefaktdaten. Die Artefaktdaten sollten bis zu diesem Schritt bereits in einer einzigen Datei zusammengeführt worden sein, sodass diese Datei mit der Moduldefinition zusammengeführt werden kann. Moduldefinitionen bestehen aus Modulen und den zugeordneten Artefakten. Da Module nicht in den Artefaktdaten enthalten sind und sich Artefakte, wie oben beschrieben, per `fullName`-Attribut zuordnen lassen, kann auch an dieser Stelle die Option `-f` bei Aufruf des Werkzeugs genutzt werden.

8.4. Datenprüfung und Anreicherung

In diesem Abschnitt werden die Werkzeuge zur Unterstützung der Teilschritte Datenprüfung und Verarbeitung der Methodik zur Durchführung artefaktbasierter Analysen beschrieben.

Wie in Abschnitt 7.5 dargestellt, wird der Schritt der Datenprüfung vor der Anreicherung durchgeführt. Die Eingabe für diesen Schritt ist eine Datei, die alle relevanten Artefaktdaten in zusammengeführter Form enthält. Bei der Prüfung werden die Artefaktdaten nicht verändert. Im Schritt der Anreicherung werden dann abgeleitete Eigenschaften berechnet und als Ausgabe eine Artefaktdatendatei mit angereicherten Artefaktdaten erstellt.

Vor der Vorstellung der Werkzeuge wird zunächst in Abschnitt 8.4.1 erläutert, warum eine Prüfung zwischen Artefaktdaten und AM sinnvoll ist. Dabei wird auf die möglichen Ursachen für eine fehlende Konsistenz eingegangen.

In den weiteren Abschnitten werden anschließend die einzelnen Werkzeuge zur Prüfung und Anreicherung von Artefaktdaten vorgestellt. Dabei ist der Schritt der Prüfung in zwei Teilschritte unterteilt: die Strukturprüfung (Abschnitt 8.4.2) und die Auswertung von OCL-Prüfbedingungen. Zur Erstellung projektspezifischer OCL-Werkzeuge für die Datenprüfung (Abschnitt 8.4.4) und Anreicherung (Abschnitt 8.4.5) kann der in Abschnitt 8.4.3 beschriebene Generator verwendet werden.

8.4.1. Fehlerarten bei der Konsistenzprüfung

Kommt es bei der Prüfung der Artefaktdaten auf Konsistenz zu dem verwendeten AM zu Fehlern, kann dies verschiedene Ursachen haben. Eine Prüfung ist in jedem Fall sinnvoll, da bei Inkonsistenzen innerhalb der Artefaktdaten ein korrektes Analyseergebnis gefährdet ist.

In Tabelle 8.20 werden die möglichen Fehler und deren Ursache aufgeführt. Dabei ist es unerheblich, ob die Inkonsistenz zwischen AM und Artefaktdaten bei der Strukturprüfung oder Prüfung der OCL-Prüfbedingungen auftritt.

Das Auftreten mehrerer Fehlerarten ist ebenfalls möglich und führt zu einer erschwerten Bestimmung der Fehlerursachen. Trotzdem lässt sich auch ein solcher Fall auf die beschriebenen Fehlerursachen zurückführen.

Darüber hinaus bleibt zu erwähnen, dass es neben einer Inkonsistenz zwischen Artefaktdaten und AM auch zu syntaktischen Fehlern sowohl in den Artefaktdaten als auch im AM kommen kann. Da diese jedoch durch die Verwendung der von MontiCore generierten Parser frühzeitig erkannt werden, werden solche Fehler an dieser Stelle nicht weiter betrachtet.

Fehler	Mögliche Ursache
Fehlerhaftes AM	Ein Fehler im AM kann vor allem durch Evolution im Projekt entstehen. Verändert sich Beispielsweise die Werkzeuglandschaft des zu analysierenden Projekts oder wird die Technologie in einem einzelnen Werkzeug ausgetauscht, kann es leicht zu neuen oder veränderten Artefakt- und Beziehungsarten kommen. In solchen Fällen ist eine Anpassung des verwendeten AMs notwendig, aber auch eine Anpassung der MontiArT-Instanz kann notwendig sein. Das Erkennen und Beheben dieser Fehlerart sichert somit die Korrektheit und die Aktualität des verwendeten AMs im Bezug auf die aktuelle Projektsituation ab.
Fehlerhafte Artefaktdaten	Fehlerhafte Artefaktdaten können durch fehlerhaft implementierte Extraktoren entstehen. Dies kann durch die Datenprüfung frühzeitig entdeckt werden. Das Erkennen und Beheben dieser Fehlerursache bildet somit eine zusätzliche Qualitätssicherungsmaßnahme für den *Analysewerkzeugentwickler*.
	Es besteht auch die Möglichkeit, dass fehlerhafte Artefaktdaten durch eine fehlerhafte Projektsituation entstehen. Manche Situationen werden durch die Modellierung des AMs kategorisch ausgeschlossen, sodass diese Situationen bei der Analyse nicht auftreten können. Ist dieser Ausschluss weiterhin gewollt, so muss das Projekt angepasst werden. Ansonsten kann durch das Anpassen des AMs die aufgetretene Projektsituation für Analysen zugelassen werden.

Tabelle 8.20.: Fehlerarten bei der Konsistenzprüfung und deren Ursachen.

8.4.2. Strukturprüfung

Die Strukturprüfung (siehe Abschnitt 7.5) ist ein Teilschritt der Konsistenzprüfung zwischen AM und zugehörigen Artefaktdaten. Dabei werden die als OD vorliegenden Artefaktdaten auf Konsistenz zu dem CD-Anteil des AMs geprüft. Im Folgenden wird das Werkzeug `CDODConsistencyChecker` vorgestellt, welches diese Prüfung automatisiert durchführt. Die Werkzeugunterstützung für den zweiten Teilschritt der Konsistenzprüfung, die Anwendung von OCL-Prüfbedingungen, wird später in Abschnitt 8.4.4 vorgestellt.

Der `CDODConsistencyChecker` wird als Teil des MontiArT-Archivs `language-tools` ausgeliefert. Das Werkzeug ermöglicht es, die Prüfung der Konsistenz zwischen einem OD und einem CD durchzuführen. Als Eingabe muss dem `CDODConsistencyChecker` genau ein CD und ein OD übergeben werden. Dabei muss das OD die in Abschnitt 5.1.2, das CD die in Abschnitt 5.1.1 beschriebenen Eigenschaften besitzen. In Listing 8.21 ist ein beispielhafter Aufruf des `CDODConsistencyCheckers` mit dem CD `AMCD.cd` und den Artefaktdaten `Data.od` dargestellt.

```
java language-tools.jar CDODConsistencyChecker AMCD.cd Data.od
```

Listing 8.21: Beispielhafter Aufruf des `CDODConsistencyCheckers`.

Neben der stets verfügbaren Hilfeoption (siehe Abschnitt 8.1.2) kann auch eine Option angegeben werden, um statt der standardmäßig angenommenen Closed World Assumption die Open World Assumption auszuwählen (vgl.Abschnitt 5.1.4). Die Option ist in Tabelle 8.22 dargestellt.

Kurz	Lang	Beschreibung
`-o`	`--openworld`	Verwendet eine Open World Assumption bei der Strukturprüfung. Dadurch werden einige der Prüfschritte wie in Abschnitt 5.1.4 beschrieben ausgelassen.

Tabelle 8.22.: Mögliche Option bei Aufruf des `CDODConsistencyCheckers`.

Das Werkzeug führt den in Abschnitt 5.1.4 beschriebenen Algorithmus aus. Da der Algorithmus bereits zur Definition der Konsistenz zwischen OD und CD genutzt wurde, ist offensichtlich, dass er sich auch zur Konsistenzprüfung eignet.

Als Rückgabestatus des CLIs wird bei erfolgreicher Prüfung *success* und bei Feststellung einer strukturellen Inkonsistenz *failure* zurückgegeben. Zusätzlich wird der Erfolg oder Misserfolg durch eine Textausgabe angezeigt. Bei Feststellung einer Inkonsistenz wird zusätzlich zu den oben beschriebenen Ausgaben ein Report erzeugt, der alle zu Inkonsistenzen führenden Stellen im OD auflistet. Dieser Report hat die Form einer Textdatei und nicht die eines ODs. Der Grund hierfür ist, dass fehlerhafte Datentypen nicht gut innerhalb einer Ausgabe im OD-Format zu erkennen wären. Darüber hinaus ist die Darstellung von zusätzlichen und fehlenden Elementen innerhalb derselben Datei schwierig. Stattdessen wird bei der zeilenweisen Auflistung der Inkonsistenzen im Report neben der Angabe des Grundes auch auf die entsprechenden Modellelemente verwiesen.

Innerhalb von MontiArT kann der `CDODConsistencyChecker` zur Strukturprüfung genutzt werden, da sowohl die Artefaktdaten als auch der CD-Anteil des AMs die Anforderungen an die Eingaben erfüllen. Der CD-Anteil des AMs lässt sich mit Hilfe des Werkzeugs `CDOCLSplitter` (siehe Abschnitt 8.6.3) separieren. Neben der Verwendung in MontiArT ist auch ein Einsatz des Werkzeugs außerhalb der artefaktbasierten Analyse denkbar. Allerdings ist darauf zu achten, dass die speziellen Eigenschaften für die Eingabemodelle erfüllt sind, was außerhalb der artefaktbasierten Analyse nicht zwingend der Fall sein muss.

8.4.3. OCL Generator

Der `OCLGenerator` ist ein Werkzeug, das nicht direkt an der Durchführung artefaktbasierter Analysen beteiligt ist. Stattdessen dient es dazu, aus OCL-Ausdrücken Werkzeuge für die

Durchführung artefaktbasierter Analysen zu generieren. Dadurch lässt sich der Aufwand zur Erstellung dieser Werkzeuge erheblich reduzieren (Anforderung *TRQ-1*). In der aktuellen Version von MontiArT liegt der Generator nur in prototypischer Form vor. Diese prototypische Version wurde bereits in [Cuj17] vorgestellt. Abbildung 8.23 gibt einen Überblick über das Vorgehen.

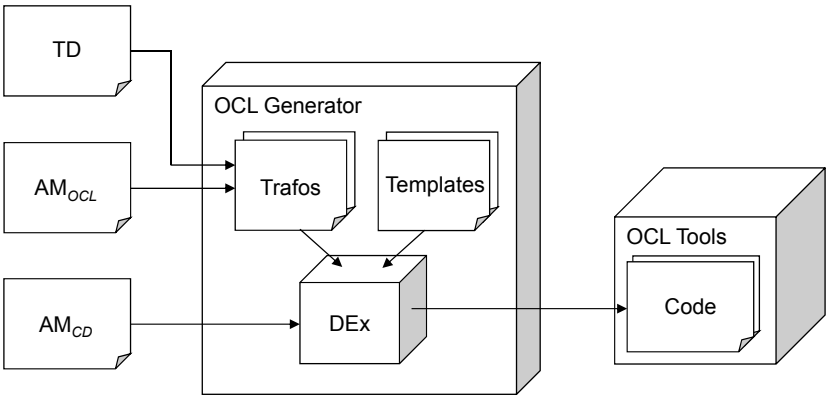

Abbildung 8.23.: Überblick über den Ablauf des `OCLGenerator`.

Der `OCLGenerator` ist ein auf DEx basierender Generator, der als Eingabe das AM (in Form der beiden Artefakte AM_{CD} und AM_{OCL}) und eine zugehörige Tagdefinition (vgl. Abschnitt 7.2.5) bekommt. Durch die Tag-Datei wird bestimmt, welche der OCL-Bedingungen aus AM_{OCL} für welchen Zweck verwendet werden sollen. Basierend auf dieser Konfiguration generiert der `OCLGenerator` das ausführbare Archiv `ocl-tools`. Die Aufteilung des AMs in zwei separate Anteile wird durch das später in Abschnitt 8.6.3 vorgestellte `Splitter`-Werkzeug ermöglicht. Durch die Trennung der Sprachanteile konnte auf die bereits existierende Sprachimplementierungen der OCL sowie der CD-Sprache zurückgegriffen werden. Tabelle 8.24 zeigt die geplanten und bereits prototypisch generierbaren Werkzeuge des Archivs, zusammen mit den verwendeten Tags der Tagdefinition.

Werkzeug	**Tag (Attribut)**	**generierbar**
`ODVerifier`	`check`	✓
`ODAccumulator`	`derive`	✓
`ODVerifier4A`	`check` (`analysis`)	✓
`ODAccumulator4A`	`derive` (`analysis`)	✓
`MetricCalculator`	`metric`	

Tabelle 8.24.: Geplante und bereits prototypisch generierbare Werkzeuge der `ocl-tools`.

Bei der bisherigen Realisierung des `OCLGenerator` enthält das von diesem Werkzeug generierte, ausführbare Archiv Werkzeuge zur Unterstützung der Schritte: Datenprüfung, Anreicherung und Analyse. Bislang fehlt eine Generierung der Werkzeuge zur Berechnung von Metriken.

Die generierten `ocl-tools` sind für einen mehrmaligen Einsatz bei einer automatisierten und wiederholbaren Durchführung der artefaktbasierten Analyse geeignet. Eine Neuerstellung des ausführbaren Archivs ist nur bei Änderung des entsprechenden AMs notwendig.

Bei dem Aufruf des `OCLGenerators` lässt sich lediglich die Hilfeoption angeben. Es wird kein Report ausgegeben, weshalb die Option zur Konfiguration des Reportverzeichnisses nicht zur Verfügung steht. Es muss wie in Listing 8.25 dargestellt neben den beschriebenen Eingabedateien das Zielverzeichnis, in welches die `ocl-tools` generiert werden, angegeben werden.

```
1 java -jar OCLGenerator AM.cd AM.ocl TD.td <outputdir>
```

Listing 8.25: Beispielhafter Aufruf des `OCLGenerators`.

Die Generierung des Codes zur Auswertung von OCL-Ausdrücken geschieht wie folgt. Für jede Klasse des AM_{CD}s wird eine Datenklasse mit den angegebenen konkreten Attributen generiert. Für Assoziationen werden ebenfalls Attribute in zu der Quellklasse der Assoziation des CDs zugehörigen Datenklassen, wie in [Rot17] beschrieben, generiert. Bei Ausführung des Werkzeugs werden die Artefaktdaten eingelesen und entsprechende Objekte der Datenklassen instanziiert, sodass die weitere Bearbeitung auf Objekten der generierten Datenklassen stattfinden kann.

Zusätzlich zu den Attributen werden für die verschiedenen Nutzungsarten Prüf- bzw. Berechnungsmethoden zu den Datenklassen hinzugefügt. Außerdem wird eine globale Prüfklasse generiert. Diese Klasse enthält alle Prüfmethoden für OCL-Prüfbedingungen, die sich nicht einer konkreten Klasse zuordnen lassen. Dies ist der Fall, wenn sich wie in Zeile 3 aus Listing 8.26 keine Kontextdefinition in der OCL-Prüfbedingungen befindet oder mehrere Kontexte wie in Zeile 6 aus Listing 8.26 angegeben sind.

```
1 OCL GlobalContext {                                          OCL
2
3   inv:
4     Artifact.size == 10;
5
6   context TextFile t, JavaSourceFile j inv:
7     t.simpleName != j.simpleName;
8 }
```

Listing 8.26: Beispiele für OCL-Prüfbedingungen, die sich keinem einzelnen Kontext zuordnen lassen.

Bei OCL-Berechnungsvorschriften muss sich der Kontext immer bestimmen lassen. Werden mehrere Klassen als Kontext angegeben, so lässt sich in der aktuellen Version des `OCLGenerators` die Zielklasse für die Methoden anhand der Form der OCL-Bedingung bestimmen

[Cuj17]. Alternativ könnten hier auch speziellere Tags verwendet werden, wodurch der Kontext oder sogar die zu berechnende Eigenschaft direkt durch ein Tag gekennzeichnet werden könnte.

Zur Realisierung der Umsetzung von Prüfbedingungen und Berechnungsvorschriften wird jeweils ein eigenes Schema zur Erstellung der Methodensignatur verwendet. Zur Realisierung beider Methodenarten müssen in beiden Fällen die einzelnen OCL-Konstrukte in Java-Quellcode überführt werden. [Cuj17] gibt eine Übersicht über die noch nicht unterstützen und bereits durch den Prototyp unterstützten Konstrukte sowie deren Transformation in Java-Code. Für eine frühere Version des in Kapitel 6 vorgestellten AMs konnten so 95% aller OCL-Spezifikationen automatisch realisiert werden.

In den Abschnitten 8.4.4 und 8.4.5 wird die Verwendung der bereits generierbaren Werkzeuge `ODVerifier` und `ODAccumulator` vorgestellt und gezeigt, wie entsprechende Java-Methoden aus OCL-Bedingungen generiert werden. In Abschnitt 8.5.2 werden dann die Werkzeuge `ODVerifier4A` und `ODAccumulator4A` beschrieben. Die automatische Umsetzung von OCL-Spezifikationen zur Metrikberechnung wird hingegen nicht weiter behandelt. Stattdessen wird in Abschnitt 8.5.5 auf die Visualisierung von Metriken eingegangen.

8.4.4. Auswertung von OCL-Prüfbedingungen

Die Auswertung von OCL-Prüfbedingungen bildet den zweiten Teil der Konsistenzprüfung zwischen den Artefaktdaten und dem AM. Dazu müssen die Artefaktdaten auf die Einhaltung der OCL-Ausdrücke geprüft werden, die in der Tagdefinition als Prüfbedingung markiert wurden. Mit dem `OCLGenerator` steht ein Werkzeug zur Generierung eines Prüfwerkzeugs, dem `ODVerifier`, zur Unterstützung dieses Schritts zur Verfügung.

Für den Schritt der Konsistenzprüfung bekommt der `ODVerifier` zu dem AM_{CD} passende Artefaktdaten als Eingabe, d.h., die Strukturprüfung wurde bereits erfolgreich durchgeführt. Als Ausgabe wird kein weiteres zu verarbeitendes Artefakt erstellt. Stattdessen wird, analog zu dem Werkzeug für die Strukturprüfung, dem `CDODConsistencyChecker`, über den Exitcode (*success* oder *failure*) angezeigt, ob die Prüfung erfolgreich war. Zusätzlich wird der Erfolg oder Misserfolg durch eine Textausgabe angezeigt. Darüber hinaus wird ein Report erzeugt, mit dessen Hilfe sich für jede fehlgeschlagene Prüfung die betreffenden Objekte sowie die zugehörige OCL-Bedingung identifizieren lassen.

Bis auf die in Abschnitt 8.1.2 dargestellten Standardoptionen können keine weiteren Optionen angegeben werden. In Listing 8.27 ist ein beispielhafter Aufruf der Konsistenzprüfung für die Artefaktdaten `Data.od` dargestellt.

```
1 java -jar ocl-tools ODVerifier Data.od
```

Listing 8.27: Beispielhafter Aufruf des `ODVerifiers`.

Bei Verwendung des `OCLGenerators` werden Prüfmethoden, wie in Abschnitt 8.4.3 beschrieben, je nach Kontext der OCL-Prüfbedingung in eine der Datenklassen oder in die globale Prüfklasse generiert. Die Methodensignatur ist in beiden Fällen gleich. Werden weitere Daten

OCL

```
context Artifact inv:
  simpleName == "";
```

Java

```
public class Artifact {
  /* ... */
  private String simpleName;

  public boolean checkOCLInvariants() {
    return simpleName.equals("");
  }
  /* ... */
}
```

Abbildung 8.28.: Überführung einer OCL-Prüfbedingung in eine Java-Prüfmethode (entnommen aus [Cuj17]).

aus anderen Datenklassen benötigt, wird dies in der aktuellen Version des `OCLGenerators` durch eine zentrale Schnittstelle ermöglicht, die in allen Datenklassen zur Verfügung steht. Eine beispielhafte Überführung einer OCL-Prüfbedingung in eine Java-Prüfmethode ist in Abbildung 8.28 dargestellt.

Die einzelnen Prüfmethoden benötigen somit keine Parameter und geben als Rückgabewert einen booleschen Wert zurück. Bei Ausführung des `ODVerifiers` werden alle Prüfbedingungen, die mit dem Tag `check` (ohne Angabe eines Attributs) markiert wurden, geprüft.

MontiArT bietet die Möglichkeit OCL-Prüfbedingungen, die durch ein `manual`-Tag markiert sind, durch handgeschriebene Werkzeuge zu realisieren. Dies hat den Vorteil, dass besonders performanter Prüfcode erstellt werden kann oder dass auch komplexe Bedingungen, die nicht für die automatische Verarbeitung geeignet sind, dennoch geprüft werden können. Nachteile sind, dass der manuelle Schritt Zusatzaufwand bedeutet, eine zusätzliche Fehlerquelle entsteht und dass die Synchronisation zwischen AM und dem handgeschriebenen Code gewährleistet sein muss. Aufgrund dieser Nachteile wird dem *Werkzeugentwickler* empfohlen, nach Möglichkeit die automatisierte Verarbeitung der OCL-Spezifikationen durch den `OCLGenerator` zu nutzen.

Bis hierhin wurde beschrieben, wie die Artefaktdaten auf Konsistenz mit dem AM geprüft werden können. Zusammen mit dem in Abschnitt 8.4.2 vorgestellten `CDODConsistencyChecker`, kann die Konsistenz der Artefaktdaten sowohl zu dem CD- als auch zu dem OCL-Anteil des AMs geprüft werden. Im folgenden Abschnitt wird nun das Konzept zur Berechnung abgeleiteter Eigenschaften für den Schritt der Anreicherung beschrieben.

OCL

```
context Artifact inv:
  name == (nameExtension == "") ? simpleName :
    simpleName + "." + nameExtension;
```

Java

```
public class Artifact {
  /* ... */
  public String getDerivedName() {
    return OCLComparator.equals(nameExtension, "") ? simpleName :
      simpleName + "." + nameExtension;
  }
  /* ... */
}
```

Abbildung 8.30.: Überführung einer OCL-Berechnungsvorschrift in eine Java-Methode zur Berechnung eines abgeleiteten Attributs (entnommen aus [Cuj17]).

8.4.5. Anreicherung der abgeleiteten Eigenschaften

Die Anreicherung von Artefaktdaten durch Berechnung abgeleiteter Eigenschaften, die als OCL-Berechnungsvorschrift spezifiziert wurden, bildet den vierten Teilschritt bei der Durchführung der Artefaktbasierten Analyse. Dazu müssen die OCL-Ausdrücke ausgewertet werden, die in der Tagdefinition mit `derive` markiert wurden. Mit dem `OCLGenerator` lässt sich der `ODAccumulator`, ein Werkzeugs zur Datenanreicherung, generieren.

Für den Schritt der Anreicherung bekommt der `ODAccumulator` zu dem AM_{CD} passende Artefaktdaten als Eingabe. Als Ausgabe wird eine OD-Datei erstellt, die die Artefaktdaten der Eingabe angereichert mit den berechneten abgeleiteten Eigenschaften enthält. Bei der Ausführung des `ODAccumulators` wird kein Report erzeugt. In Listing 8.29 ist ein beispielhafter Aufruf des Werkzeugs für die Eingabeartefaktdaten `Data.od` dargestellt.

```
java -jar ocl-tools ODAccumulator Data.od EnrichedData.od
```

Listing 8.29: Beispielhafter Aufruf des `ODAccumulators`.

Neben der Auswahl des Werkzeugs und der Eingabedatei muss hier auch die Ausgabedatei (hier `EnrichedData.od`) angegeben werden. Bei Aufruf des Werkzeugs können außer der Option zur Darstellung des Hilfetextes keine weiteren Optionen angegeben werden.

Bei Verwendung des `OCLGenerators` werden Methoden zur Berechnung abgeleiteter Eigenschaften aus den entsprechenden OCL-Berechnungsvorschriften generiert. Da solche Ausdrücke immer einen Kontext haben müssen, damit sie sich eindeutig Berechnen lassen (siehe Abschnitt 5.1.3), werden die Methoden in der zugehörigen Datenklasse erstellt. Eine beispielhafte Überführung einer OCL-Berechnungsvorschrift zur Berechnung eines abgeleiteten Attributs in eine Java-Methode ist in Abbildung 8.30 dargestellt.

Die Anreicherungsmethode gibt das Java-Pendant der berechneten Eigenschaft als Wert zurück. Durch die Verwendung der gleichen Datenstrukturen wie der des `ODVerifiers` ist die Übergabe von Parametern nicht nötig. Das dargestellte Vorgehen bei der Generierung hat den Vorteil, dass sich abgeleitete Eigenschaften aus anderen abgeleiteten Eigenschaften berechnen lassen, da die Berechnung dynamisch, während des Aufrufs der Methode stattfindet. Eine bestimmte Reihenfolge der Berechnung muss somit nicht beachtet werden. Bei Ausführung des `ODAccumulators` werden alle OCL-Ausdrücke, die mit dem Tag `derive` (ohne Angabe eines Attributs) markiert wurden, ausgewertet. Die berechneten Eigenschaften werden den Artefaktdaten hinzugefügt.

Bei der vorgeschlagenen Reihenfolge von Konsistenzprüfung und Anreicherung können auch abgeleitete Eigenschaften zur Konsistenzprüfung verwendet werden. Durch die Verwendung gemeinsamer Datenklassen und die Realisierung der Berechnung abgeleiteter Eigenschaften als Methoden lassen sich diese Methoden auch bereits bei der Prüfung von Konsistenzbedingungen nutzen.

8.5. Datenanalyse

In diesem Abschnitt wird die tooltechnische Unterstützung des fünften und letzten Schritts der Methodik zur Durchführung artefaktbasierter Analysen von MDD-Projekten beschrieben: die Artefaktdatenanalyse. Als Eingabe für diesen Schritt werden geprüfte und angereicherte Artefaktdaten verwendet. Zur Durchführung der verschiedenen in Abschnitt 7.6 vorgestellten Analysemöglichkeiten bietet MontiArT unterschiedliche Werkzeuge an.

In Abschnitt 8.5.1 wird ein Werkzeug zur Datenexploration vorgestellt. Die Beschreibung der Werkzeuge für die OCL-basierte Analyse, die Filterung von ODs und die Architekturanalyse folgen in den Abschnitten 8.5.2 bis 8.5.4. Außerdem wird in Abschnitt 8.5.5 ein Werkzeug zur Visualisierung von Artefaktdaten vorgestellt.

8.5.1. Artifact Data Explorer

Der *Artifact Data Explorer* (ArtEx) ist ein mit dem DEx-Generator [Rot17] erstelltes Werkzeug, welches ein komfortables Explorieren der Artefaktdaten erlaubt. Somit trägt das Werkzeug zur Erfüllung von Anforderung *TRQ-4.1* bei. Daten werden im *Artifact Data Explorer* (ArtEx) in Form von Tabellen und Formularfeldern dargestellt, d.h. es wird die durch den DEx-Generator automatisch bereitgestellte GUI zur Exploration der Daten verwendet [Rot17]. Die Erstellung des Werkzeugs erfolgt generativ. Als Eingabe bekommt der DEx-Generator den CD-Anteil AM_{CD} des AMs. Als Ergebnis wird direkt das Werkzeug ArtEx ausgegeben. Nach dem Starten des Werkzeugs lassen sich Artefaktdaten einfach per Menüeintrag importieren.

Der DEx-Generator generiert zur Berechnung abgeleiteter Attribute, ähnlich wie der `OCL-Generator` (vgl. Abschnitt 8.4.3), eigene Methoden [Rot17]. Die Implementierung dieser Methoden ist aber, im Unterschied zu dem `OCLGenerator`, nur durch handgeschriebenen

Code möglich. Da die abgeleiteten Attribute in den zu explorierenden Daten bereits hinzugefügt wurden, ist die erneute Berechnung der Werte unnötig. Stattdessen können durch eine Vortransformation die bereits berechneten abgeleiteten Attribute aus AM_{CD} in normale Attribute umgewandelt werden. Da abgeleitete Attribute im Gegensatz zu normalen Attributen nicht verändert werden dürfen, ist eine solche Transformation normalerweise als kritisch anzusehen. Für ArtEx ist dies jedoch nicht der Fall, da das Werkzeug nur zum Explorieren, nicht aber zum Ändern von Artefaktdaten genutzt wird.

Die Spezifikation des AMs in Form eines CDs und die Erhebung der Daten im OD-Format haben an dieser Stelle einen großen Vorteil mit sich gebracht. Der Aufwand für die Erstellung von ArtEx in der beschriebenen Form ist sehr gering, da die Erstellung vollständig automatisiert werden kann, was zur Erfüllung von Punkt *TRQ-1* beiträgt. Dies ist vor allem bei solchen Änderungen im zu untersuchenden Projekt von Vorteil, die eine Änderung des zugehörigen AMs nach sich zieht. Eine Anpassung von ArtEx auf die neue AM-Version ist durch eine automatisierte Neuerstellung des Werkzeugs ohne zusätzlichen Aufwand möglich.

8.5.2. OCL-basierte Analysen

Bei der Durchführung OCL-basierter Analysen werden Analysenspezifikationen, wie die in Abschnitt 7.2.1 beschrieben und in Abschnitt 7.6 beispielhaft durchgeführten, berechnet. MontiArT bietet mit dem `OCLGenerator` (siehe Abschnitt 8.4.3) einen Generator zur automatisierten Erstellung von Werkzeugen, die solche Analysen durchführen können.

Die Generierung und Ausführung der Werkzeuge zur Durchführung der OCL-basierten Analyse geschieht analog zu der in den Abschnitten 8.4.4 und 8.4.5 beschriebenen Generierung. Kleinere Unterschiede bestehen lediglich darin, dass der entsprechende Werkzeugname bei Aufruf des ausführbaren Jar-Archivs ausgetauscht werden muss (`ODVerifier4A` statt `ODVerifier` bzw. `ODAccumulator4A` statt `ODAccumulator`). Außerdem werden bei der Ausführung der Werkzeuge andere OCL-Ausdrücke ausgewertet. So werden bei Ausführung des `ODVerifier4As` alle Prüfbedingungen, die sowohl mit dem Tag `check` als auch mit dem Tag `analysis` markiert wurden, geprüft. Analog dazu werden bei Ausführung des `ODAccumulator4As` alle Berechnungsvorschriften ausgewertet, die sowohl mit `derive` als auch mit `analysis` getaggt wurden.

Der technische Unterschied zwischen den Werkzeugen zur Analyse und denen zur Prüfung bzw. Anreicherung ist praktisch nicht vorhanden. Allerdings ist eine Trennung der Abläufe bei der Durchführung der artefaktbasierten Analyse wichtig, um sie zu verschiedenen Zeitpunkten durchführen und um auf ein negatives Prüfergebnis unterschiedlich reagieren zu können. Bei einem negativen Ergebnis während der Konsistenzprüfung mit dem `ODVerifier` soll die Durchführung der artefaktbasierten Analyse abgebrochen werden, während dies bei der Analyse mit dem `ODVerifier4A` nicht der Fall ist. Eine solche Reaktion auf Analyseergebnisse geschieht in MontiArT allerdings außerhalb der Werkzeuge. Wie bei Verwendung eines Shell-Skripts zur Konfiguration der MontiArT-Instanz unterschiedlich auf die jeweiligen Ergebnisse

reagiert werden kann, wird in Abschnitt 9.3 beschrieben. Durch die Aufteilung in verschiedene Werkzeuge wird auch eine Trennung der Abläufe realisiert.

8.5.3. OD Filter

Der `ODFilter` führt bei Ausführung eine Filterung entsprechend der in Abschnitt 7.6 vorgestellten Filterung durch. Dazu bekommt er ein OD als Eingabe und produziert ein OD als Ausgabe.

Konzeptionell würde der `ODFilter` sich gut in die `language-tools` einfügen, da er die Filterung unabhängig vom verwendeten AM auf beliebigen ODs durchführen kann. Da der `ODFilter` jedoch anders als die Werkzeuge der `language-tools` bereits auf der aktuellen Sprachversion der OD-Sprache arbeitet, wird er in einem eigenen ausführbaren Jar-Archiv ausgeliefert. Nach einer Migration der `language-tools` auf die neue OD-Version kann er dort integriert werden. Ein beispielhafter Aufruf des `ODFilters` ist in Listing 8.31 dargestellt.

```
1 java od-filter -o "JavaSourceFile, FreeMarkerTemplate" \
2 -l "contributionFT, contributionJava" Input.od Output.od
```

Listing 8.31: Beispielhafter Aufruf des `ODFilters`.

Die Objekttypen und Linknamen, nach denen gefiltert werden soll, können, wie in in Tabelle 8.32 beschrieben, durch einen kommaseparierten String bei Aufruf des Werkzeugs als Option angegeben werden. Standardmäßig werden alle Objekte und Links übertragen. Neben den dargestellten Optionen ist ansonsten nur der Aufruf des Hilfetexts möglich.

Kurz	Lang	Beschreibung
`-o`	`--outputtypes`	Angabe der zulässigen Objekttypen. Nur Objekte der angegebenen Typen werden in das Ausgabe-OD übernommen. Wird dieser Parameter nicht genutzt, sind alle Objekttypen zulässig.
`-l`	`--linktypes`	Angabe der zulässigen Linknamen. Nur Links mit den angegeben Namen werden in das Ausgabe-OD übernommen. Wird dieser Parameter nicht genutzt, sind alle Linknamen zulässig.

Tabelle 8.32.: Mögliche Optionen bei Aufruf des `ODFilters`.

Für den in Listing 8.31 dargestellten Aufruf enthält das Ausgabe OD `Output.od` nur Objekte der Typen `JavaSourceFile` und `FreeMarkerTemplate`. Darüber hinaus sind in der Ausgabe keine Links enthalten, deren Namen nicht `contributionFT` oder `contributionJava` sind.

Bei der Prüfung, ob ein Link übernommen wird oder nicht, wird in der aktuellen Version des Tools nur der Name des Links geprüft. Da ein Assoziationsname in der CD4A-Sprache nicht zwingend eindeutig sein muss, könnten so auch unerwünschte Links übernommen werden. Aus diesem Grund wird zusätzlich geprüft, ob die durch den Link verknüpften Objekte zu

zulässigen Objekttypen gehören. Ist dies für mindestens ein Objekt nicht der Fall, wird der entsprechende Link nicht übernommen. Auch durch diese Regelung kann es zur unerwünschten Übernahme von Links kommen, wenn auch die verbundenen Objekte des unerwünschten Links übernommen werden. Bei der Weiterentwicklung des Werkzeugs sollte deshalb eine Möglichkeit geschaffen werden, Assoziationen eindeutig zu referenziert.

Eine weitere zukünftige Erweiterung für das Werkzeug stellt die Unterstützung von hierarchischen Links dar. Obwohl bei der Implementierung bereits Version 2.2.0 der OD-Sprache verwendet wurde, werden hierarchische Links aktuell als Eigenschaft des zugehörigen Objekts angesehen und sind somit genau dann in der Ausgabe enthalten, wenn auch das Objekt enthalten ist. Normale Links von oder zu hierarchisch definierten Objekten werden hingegen nicht der Ausgabe hinzugefügt, auch wenn der zugehörige Linkname passend ist. Durch eine Vortransformation mit dem `ODFlattener` (vgl. Abschnitt 8.6.1) lassen sich hierarchisch verschachtelte Objekte und Links transformieren, wodurch sie bei der Filterung dann ebenfalls berücksichtigt werden können.

Der `ODFilter` kann sowohl für Analysezwecke als auch für die Vorbereitung der Analyse verwendet werden. Bei der Analyse lassen sich einzelne Analyseergebnisse auf die relevanten Objekt- und Linktypen einschränken. Sind bestimmte extrahierte Objekt- und Linktypen für keine der Analysen relevant, lassen sich diese bereits in einem früheren Schritt bei der Durchführung der artefaktbasierten Analyse (z.B. nach der Extraktion) entfernen, wodurch bereits die Zwischenergebnisse auf die relevanten Objekt- und Linktypen reduziert werden können. Neben diesen Anwendungen in MontiArT ist auch eine Verwendung außerhalb der artefaktbasierten Analyse denkbar.

8.5.4. Architekturanalyse

In diesem Abschnitt wird beschrieben, wie sich die in Abschnitt 7.6 vorgestellte Architekturanalyse mit Hilfe von MontiArT durchführen lässt. Das dort beschriebene Vorgehen zur Berechnung einer Differenzarchitektur lässt sich durch die folgenden Schritte zusammenfassen:

1. Modellierung der Soll-Architektur.
2. Modellierung der Moduldefinition.
3. Berechnung der Ist-Architektur.
4. Berechnung der Differenzarchitektur aus der Soll- und Ist-Architektur.

Die Modellierung der Soll-Architektur und der Moduldefinition sind manuelle Schritte, die von dem *Analysten* durchgeführt werden müssen (siehe Abschnitt 7.2.3). Die weiteren beiden Schritte lassen sich jedoch durch Werkzeuge unterstützen. Im Folgenden werden die beiden Werkzeuge zur Durchführung der Schritte (3) und (4) vorgestellt.

Architecture Reconstructor Der `ArcReconstructor` führt Schritt (3) der oben beschriebenen Schritte durch, d.h., er berechnet aus den zusammengeführten Daten aus übergebenen Artefaktdaten und einer zugehörigen Moduldefinition die Ist-Architektur. Wie in Abschnitt 7.2.3 beschrieben, ist aus Komfortgründen eine Zuweisung ganzer Ordner zu Modulen bei der Spezifikation der Moduldefinition möglich. Aus diesem Grund lässt sich der in Abschnitt 8.5.2 beschriebenen `ODAccumulator4A` nicht nutzen. Stattdessen wurde ein eigenes Werkzeug für diesen Arbeitsschritt entwickelt.

Der `ArcReconstructor` wird zusammen mit weiteren AM-spezifischen Analysewerkzeugen in einem gemeinsamen ausführbaren Jar-Archiv, den `analysis-tools`, ausgeliefert. Listing 8.33 zeigt einen beispielhaften Aufruf des Werkzeugs.

```
1 java analysis-tools.jar ArcReconstructor ModuleDataMerge.od \
2 IstArc.od
```

Listing 8.33: Beispielhafter Aufruf des `ArcReconstructors`.

Bei dem Aufruf des `ArcReconstructors` wird ein Report erzeugt, der angibt, welche Artefaktbeziehungen der Artefaktdaten zu einer Modulbeziehung in der Ist-Architektur geführt haben. Entsprechend können die beiden Standardoptionen (siehe Abschnitt 8.1.2) bei Aufruf des Werkzeugs angegeben werden.

Bei der Architekturrekonstruktion wird die in Abschnitt 6.6.1 spezifizierte Analyse durchgeführt. Dabei wird zwischen zwei verschiedenen Arten von Modulbeziehungen unterschieden: `reliesOnModule` und `contributionModule`. Während `reliesOnModule`-Beziehungen bei Vorhandensein von `refersTo`-Beziehungen zwischen entsprechenden Artefakten erstellt werden, sind für die `contributionModule`-Beziehung `produces`-Beziehungen zwischen Artefakten verantwortlich. Dabei müssen die abgeleiteten Artefaktbeziehungen `refersTo` und `produces` nicht zwingend vorher berechnet werden, da der `ArcReconstructor` auch direkt auf die Anwesenheit der zugehörigen, spezialisierten Beziehungen des AMs (vgl. Kapitel 6) reagiert.

Architecture Consistency Checker Der `ACChecker` führt Schritt (4) der oben beschriebenen Schritte durch, d.h., er berechnet aus der modellierten Soll-Architektur und der zuvor berechneten Ist-Architektur die Differenzarchitektur. An dieser Stelle könnte auch auf den `ODAccumulator4A` zurückgegriffen werden. Dazu müssten beiden Eingabeartefakte zuvor mit Hilfe des `ODMergers` in einen gemeinsamen Datensatz zusammengeführt werden. Da es zu Kommunikationszwecken vorteilhaft sein kann, dass nur die Unterschiede zwischen Soll- und Ist-Architektur in einem separaten Datensatz vorliegen, wurde an dieser Stelle ebenfalls ein eigenes Werkzeug entwickelt, das, wie auch der `ArcReconstructor`, in dem ausführbaren Jar-Archiv `analysis-tools` ausgeliefert wird. Im Unterschied zu einer Nutzung des `ODAccumulator4As` sind in der Ausgabe nur die Unterschiede zwischen Soll- und Ist-Architektur enthalten. Der `ACChecker` lässt sich wie in Listing 8.34 dargestellt aufrufen.

```
java analysis-tools.jar ACChecker SollArc.od IstArc.od DiffArc.od
```

Listing 8.34: Beispielhafter Aufruf des `ACCheckers`.

Da kein Report erstellt wird und auch sonst keine weiteren Optionen zur Verfügung stehen, kann zusätzlich nur der Hilfetext per Option aufgerufen werden.

Zur Berechnung der Differenzarchitektur führt der `ACChecker`, die in Abschnitt 6.6.2 spezifizierte Analyse durch. Soll- und Ist-Architektur werden dabei von dem Werkzeug als integrierter Datensatz behandelt. In der Ausgabedatei sind ausschließlich die Links zu den Assoziationen `unintendedReference`, `unintendedContribution`, `missingReference` und `missingContribution` enthalten.

Für jeden dieser Links l werden beide, zu l verlinkten Objekte der Ausgabe hinzugefügt, sofern sie dort noch nicht enthalten sind. Ein direkter Vergleich zwischen den Objekten der Soll- und Ist-Architektur wird von dem `ACChecker` nicht durchgeführt. Dies ist aber auch nicht notwendig, da es sich bei den Eingabe-ODs um valide Objektdiagramme handeln muss. Dadurch ist gewährleistet, dass zu jedem Link die entsprechenden Objekte vorhanden sind. Somit würde bei einer unterschiedlichen Anzahl an Modulen ein Link zu oder von einem Modul, welches nur in einer der Dateien vorhanden ist, zum Hinzufügen dieses Moduls in der Ausgabe führen, da ein entsprechender Link in dem anderen OD nicht existieren kann. Die einzige Ausnahme bilden Objekte, die keine Beziehung zu anderen Objekten haben und nur in einem der ODs vorkommen. Solche Objekte werden in der aktuellen Form nicht der Ausgabe hinzugefügt, sind für die Architekturanalyse aber auch wenig interessant.

[The17] beschreibt eine Erweiterungen der Applikation ArtViz (siehe Abschnitt 8.5.5) mit deren Hilfe sich Differenzarchitekturen auch graphisch darstellen lassen. Darüber hinaus ermöglicht diese Erweiterung eine graphische Modellierung von Soll-Architektur und Moduldefinition. Eine Unterscheidung zwischen verschiedenen Beziehungsarten, wie in diesem Abschnitt beschrieben, wird dort zum Zeitpunkt der Erstellung dieser Arbeit allerdings nicht unterstützt.

8.5.5. Visualisierung von Artefaktdaten

In diesem Abschnitt wird die Visualisierung von Artefaktdaten mit Hilfe der *Artifact Visualization* (ArtViz) vorgestellt. ArtViz ist eine Webanwendung deren Konzept bereits in [GLR17] veröffentlicht wurde. Im Folgenden wird auf die Funktionsweise der Anwendung und die Verwendung durch einen *Analysten* eingegangen. Demgegenüber wird die Konfiguration von ArtViz durch einen *Administrator* später in Abschnitt 9.4 beschrieben. Die Konfiguration erlaubt sowohl die Verwaltung mehrerer als auch die Konfigurationen einzelner Projekte.

ArtViz arbeitet mit dem OD-Format, d.h., bei Nutzung von MontiArT lassen sich beispielsweise Extraktionsergebnisse, angereicherte Artefaktdaten oder im OD-Format vorliegende Analyseergebnisse visualisieren. Darüber hinaus wurde eine zusätzliche Ansicht zur Darstellung von Metriken implementiert. Neben den Daten im OD-Format benötigen die verschiedenen Vi-

sualisierungsansichten die Daten in einem *JavaScript Object Notation* (JSON) -Format [Int17b]. Dieses Format lässt sich mit dem Konverterwerkzeug `OD2JSON` (siehe Abschnitt 8.6.2) aus ODs erzeugen. Die Daten müssen in der aktuellen Version von ArtViz von einem *Administrator* an eine vordefinierte Stelle auf dem zugehörigen Webserver abgelegt werden. Dieser Vorgang lässt sich jedoch, wie in Abschnitt 9.5 beschrieben, automatisieren.

ArtViz besteht aus drei Ansichten: *Textansicht*, *Graphansicht* und *Metrikansicht*, zwischen denen per Menü gewechselt werden kann. Für die textuelle Ansicht ist es notwendig, dass die Daten im OD-Format vorliegen. Für die anderen beiden Ansichten werden hingegen Daten im JSON-Format benötigt. Dem Nutzer der Webanwendung bleiben diese technischen Details verborgen, d.h., das JSON-Format ist für den Nutzer an keiner Stelle sichtbar.

Bevor die einzelnen Ansichten von ArtViz beschrieben werden, wird zunächst auf die Menüleiste der Anwendung eingegangen. Diese befindet sich auf der linken Seite des oberen Rands der Applikation und ist in Abbildung 8.35 dargestellt. Die Menüleiste hat bei allen Ansichten den gleichen Inhalt.

Abbildung 8.35.: Links: Die ArtViz Menüleiste; rechts: Der Zeitpunkt der Datenerstellung des ausgewählten Projekts.

Durch die *Ansichtsauswahl* lässt sich zwischen den verschiedenen Ansichten navigieren. Mittels *Projektauswahl* kann zwischen den verschiedenen, verwalteten Projekten gewechselt werden. Die *Datenauswahl* ermöglicht es zwischen den zu einem Projekt zugehörigen Datensätzen zu wechseln. Durch Betätigung des *Home*-Buttons lässt sich die Webseite in ihren Initialzustand zurückversetzen, d.h., es wird die textuelle Ansicht mit dem Standardprojekt und Standarddatensatz angezeigt. Diese Einstellungen lassen sich, wie später in Abschnitt 9.4 beschrieben, durch den *Administrator* konfigurieren. Im Folgenden werden die verschiedenen Ansichten, beginnend mit der Textansicht, vorgestellt. Der *Zeitpunkt der Datenerstellung* ist ebenfalls in Abbildung 8.35 abgebildet. Er wird auf der rechten Seite des oberen Randes der Applikation angezeigt. Dabei wird das älteste Datum aller vorhandenen Datensätze des aktuellen Projekts verwendet.

Mit Hilfe der Textansicht, lassen sich Textinhalte von Dateien anzeigen. Dies schließt insbesondere Dateien, die in Form von ODs vorliegen, mit ein. Auf der linken Seite lassen sich über eine Baumstruktur die verschiedenen Datensätze auswählen. Die *Datenauswahl* per Menü (Abbildung 8.36) aktualisiert dabei die Auswahl der Baumstruktur und umgekehrt. Auf der rechten Seite wird jeweils der ausgewählte Datensatz in textueller Form angezeigt. Dieser lässt sich auch per *Download*-Button herunterladen.

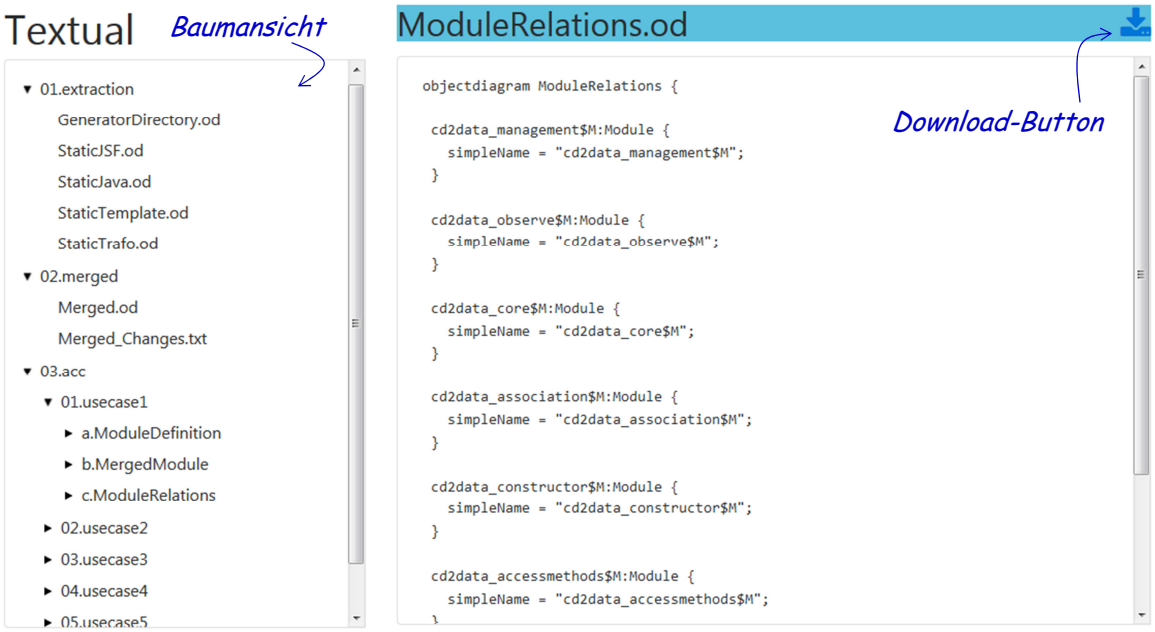

Abbildung 8.36.: Textuelle Darstellung von Artefaktdaten durch die Textansicht von ArtViz.

Die zweite Ansicht ist die Graphansicht. Eine erste Version dieser Ansicht wurde in [Ded16] entwickelt. Wie in Abbildung 8.37 dargestellt, besteht auch die Graphansicht aus einer Baumstruktur auf der linken und einer Ansicht zur Darstellung der Auswahl auf der rechten Seite.

Im Gegensatz zur Textansicht wird in der Baumstruktur der Graphansicht der Inhalt eines Datensatzes verwendet. Dabei werden Artefaktcontainer und die beinhaltenden Artefakte (vgl. Abschnitt 5.2.1) dargestellt. Die Auswahl eines Artefakts aus der Baumdarstellung aktualisiert die Ansicht auf der rechten Seite. Dort wird zwischen zwei verschiedenen Unteransichten unterschieden: *flache Graphvisualisierung* und *hierarchische Graphvisualisierung*. Die Auswahl der Unteransicht lässt sich über Auswahl der Tabs *Flat* und *Hierarchy* steuern. Alle in der Graphansicht angezeigten Elemente verwenden das Attribute `simpleName` zur Anzeige. Ist ein solches Attribut nicht vorhanden, wird das `name`-Attribut verwendet. Existiert für ein Objekt auch dieses Attribut nicht, wird stattdessen der Objektidentifikator genutzt.

Die flache Graphvisualisierung wird im Folgenden zuerst vorgestellt. Sie wird initial dargestellt, jedoch wird erst nach Auswahl eines Elements ein Graph angezeigt. Mit dieser Unteransicht können alle in den Artefaktdaten beinhalteten Elemente dargestellt werden (nicht nur die Artefakte). Elemente werden als Knoten $v \in V$, Beziehungen zwischen Elementen als Kanten $E \subseteq V \times V$ eines gerichteten Graphen $G = (V, E)$ dargestellt [Die06]. Per Rechtsklick auf einen Knoten oder eine Kante lassen sich Details der zugehörigen Artefaktdaten einblenden. Wie in Abbildung 8.38 gezeigt, werden dabei für Knoten die Daten des Objekts im OD-Format angezeigt, während für eine Kante Typ und Anzahl der vorhandenen Beziehungen dargestellt werden.

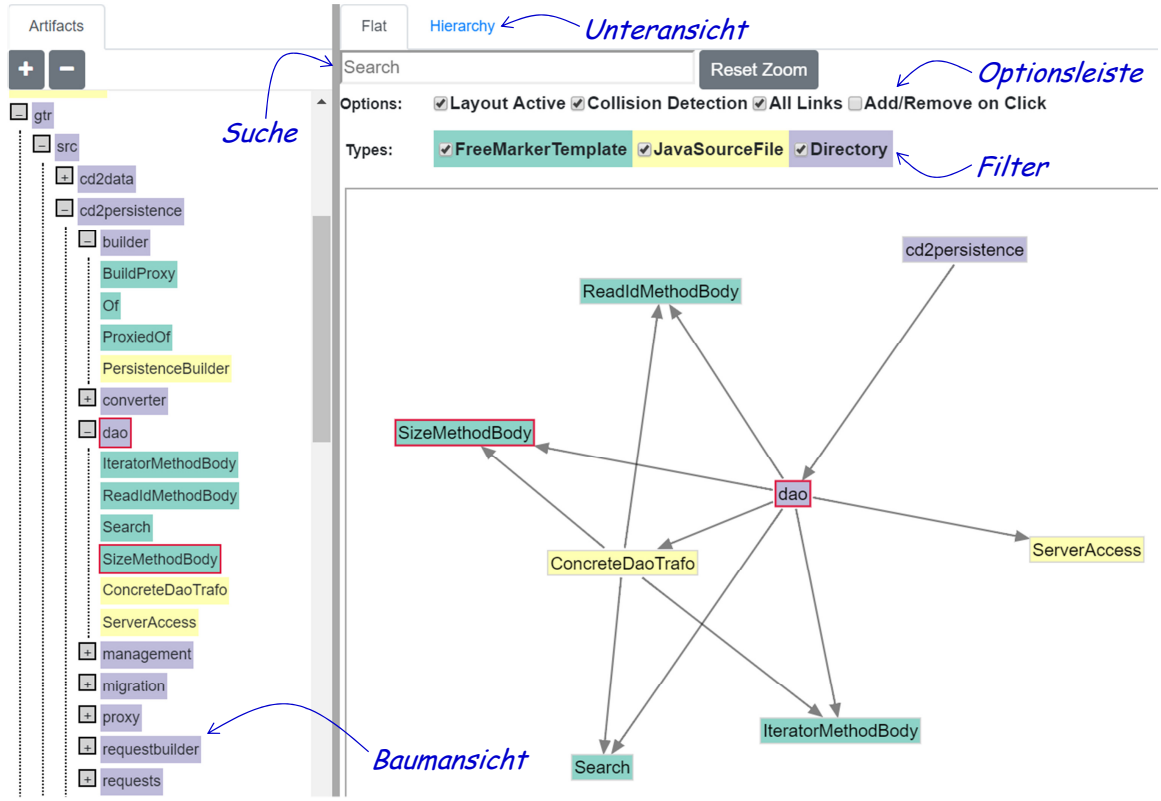

Abbildung 8.37.: Graph Visualisierung von Artefaktdaten mit ArtViz.

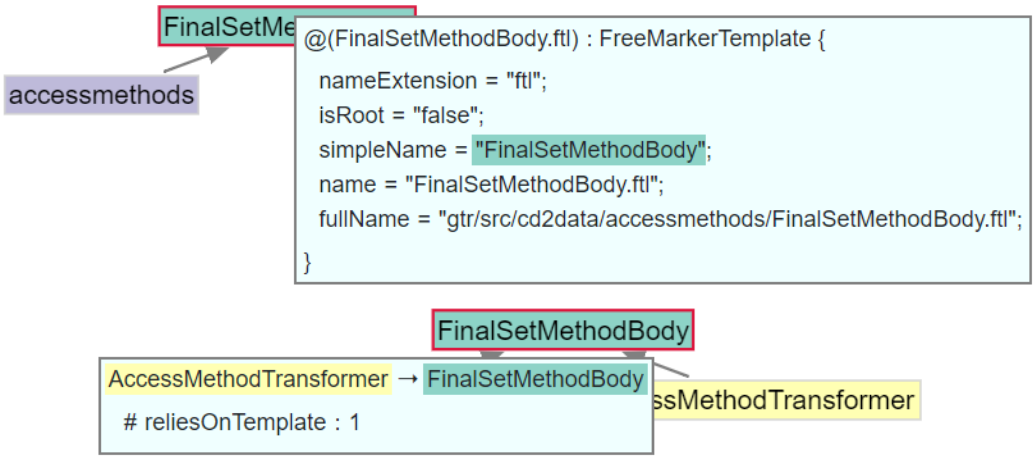

Abbildung 8.38.: Oben: Details eines Knotens; unten: Details einer Kante.

Die Darstellung als Graph ermöglicht ein Navigieren durch die Artefaktdaten, da sich die Menge der dargestellten Elemente aus der Menge der aktuell auswählten Knoten V_s bestimmt, die sich interaktiv verändern lässt. Die Auswahl einer Kante wird dabei als Auswahl der verbundenen Knoten interpretiert. Der dargestellte Teilgraph $G' = (V', E')$ mit $V' \in V$ und $E' \in E$ wird wie folgt berechnet:

- $V' = V_s \cup \{v \in V \mid a \in V_s, (a, v) \in E \vee (v, a) \in E\}$
- $E_s := \{(a, b) \in E \mid a \in V_s \vee b \in V_s\}$ oder
- $E_s := \{(a, b) \in E \mid a \in V' \wedge b \in V'\}$, falls die Option *all links* aktiviert ist.

Die Menge der dargestellten Knoten V' entspricht somit der Menge der ausgewählten Knoten V_s vereinigt mit allen Vorgängern und Nachfolgern (*inverse correspondence* Γ^{-1} bzw. *correspondence* Γ nach [Chr75]) der ausgewählten Knoten. Abhängig von der Option *All Links* werden alle Kanten zwischen den dargestellten Knoten angezeigt oder nur solche, die einen Knoten aus V_s als Quell- oder Zielknoten haben. Diese Option sowie einige andere Optionen lassen sich über die *Optionsleiste* anwählen. Eine Liste der Optionen und deren Auswirkung findet sich in Tabelle 8.39.

Option	**Auswirkung**
Layout Active	Bewirkt, dass der Layoutalgorithmus die Ausrichtung der dargestellten Elemente übernimmt. Ohne diese Option lassen sich die Elemente manuell anordnen.
Collision Detection	Bewirkt, dass Knoten nicht überlappend dargestellt werden können. Diese Option kann nur dann deaktiviert werden, wenn *Layout Active* ebenfalls deaktiviert wurde.
All Links	Bei Anwahl werden alle Kanten zwischen den dargestellten Knoten angezeigt. Ansonsten werden nur die Kanten angezeigt, die einen Knoten aus V_s als Quell- oder Zielknoten haben.
Add/Remove on Click	Bei Anwahl dieser Option werden ausgewählte Knoten zu V_s hinzugefügt. Wird ein Knoten ausgewählt der sich bereits in V_s befindet, so wird dieser entfernt. Ohne diese Option werden bei Auswahl eines Knotens zunächst alle Knoten aus V_s entfernt, bevor der ausgewählte Knoten hinzugefügt wird.
Types	In V_s befinden sich nur die hier angewählte Typen. Bei einer Änderung wird V_s entsprechen aktualisiert.

Tabelle 8.39.: Liste der mittels *Optionsleiste* wählbaren Einstellungen der *Graphansicht* von ArtViz.

Weiterhin gibt es die Möglichkeit, nach Elementen in den Artefaktdaten zu suchen und per *Zoom*-Button die Zoomstufe auf ihren Anfangswert zurückzusetzen. Ein Zoomen ist mittels Mausrad möglich.

Die flache Graphvisualisierung erlaubt es dem *Analysten*, die Beziehungen in dem untersuchten Projekt zu visualisieren, sie zu verstehen und zwischen ihnen zu navigieren, wodurch eine Möglichkeit zur komfortablen Exploration der Artefaktdaten geschaffen wurde. Dies trägt zur Erfüllung der Anforderung *TRQ-4.1* bei. Bei dem Entwurf dieser Ansicht wurde sich an dem *Massey Architecture Explorer* [Die12] orientiert. Dieser eignet sich dazu, die Struktur von Java Programmen zu analysieren. Ein anderes Werkzeug, das eine ähnliche Ansicht für Beziehungen im Quellcode von GPLs unterstützt ist Visual Studio [Vis17] mit seinem Feature *Code Maps*.

Die hierarchische Graphvisualisierung ist ebenfalls Teil von ArtViz und wurde als Unteransicht implementiert. Sie zeigt, anders als die flache Graphvisualisierung, nicht alle Elemente des ausgewählten Datensatzes an. Stattdessen werden nur ausgewählte Beziehungen visualisiert, von denen einige hierarchisch dargestellt werden. Welche Beziehungen wie dargestellt werden, lässt sich pro Beziehungsart durch den *Administrator* festlegen (siehe Abschnitt 9.4). Abbildung 8.40 zeigt die hierarchische Verschachtlung von Elementen in der hierarchischen Ansicht. In diesem Beispiel wurde die Organisation von Artefakten im Dateisystem visualisiert. Die dargestellten Details der Kante zeigen alle zu dieser Kante zugehörigen Beziehungen an.

Wie in Abschnitt 3.7.4 beschrieben, lassen sich bei dieser Art der Visualisierung Oberelemente expandieren, wodurch die enthaltenen Elemente ebenfalls dargestellt werden. Beziehungen werden immer nur für die untersten der aktuell dargestellten Elemente des Hierarchiebaums angezeigt. Wie ebenfalls in Abschnitt 3.7.4 beschrieben, lässt sich diese Art der Visualisierung besonders gut bei der Architekturanalyse verwenden. In MontiArT kann sowohl die Ist- als auch die Soll-Architektur durch die hierarchische Graphvisualisierung dargestellt werden. Durch die gegebenen Konfigurationsmöglichkeiten ist es aber auch möglich, andere Elemente der Artefaktdaten hierarchisch verschachtelt zu visualisieren. Kandidaten für eine solche Visualisierung sind die Paketstruktur von Java Programmen oder, wie in Abbildung 8.40 gezeigt, die Organisation der Artefakte im Dateisystem. In den Details zu der dargestellten Kante wird vor den drei Beziehungsarten ein '*' angezeigt. Dies bedeutet, dass es sich um Beziehungen zwischen beinhalteten Knoten handelt.

Die *Metrikansicht* ist die dritte Ansicht von ArtViz. Mit Hilfe dieser Ansicht lassen sich berechnete Metrikmaßzahlen auf verschiedene Arten darstellen. Die einfachste Darstellungsform ist dabei die Darstellung als Tabelle. Wie in Abbildung 8.41 gezeigt, wird dabei pro berechneter Metrik eine Spalte und pro Element aus den Artefaktdaten eine Zeile dargestellt.

Lässt sich eine Normierung der Metrik vornehmen, so können Maßzahlen, wie in der rechten Spalte von Abbildung 8.41, durch eine Farbcodierung hervorgehoben werden. Darüber hinaus wurden verschiedenen Möglichkeiten geschaffen, berechnete Maßzahlen in Diagrammen darzustellen.

Abbildung 8.42 zeigt die Darstellung von Metrikmaßzahlen als *Kuchendiagramm*. Die darzustellende Metrik kann per Dropdownliste ausgewählt werden. In den Metrikdiagrammen werden jeweils die ersten ersten n Elemente geordnet nach ihren Maßzahlen dargestellt, wobei sich n über ein Auswahlfeld einstellen lässt. Darüber hinaus lässt sich diese Beschränkung per Option abwählen. Außerdem kann zwischen den verschiedenen Diagrammen zur Darstellung einer Metrik interaktiv gewechselt werden.

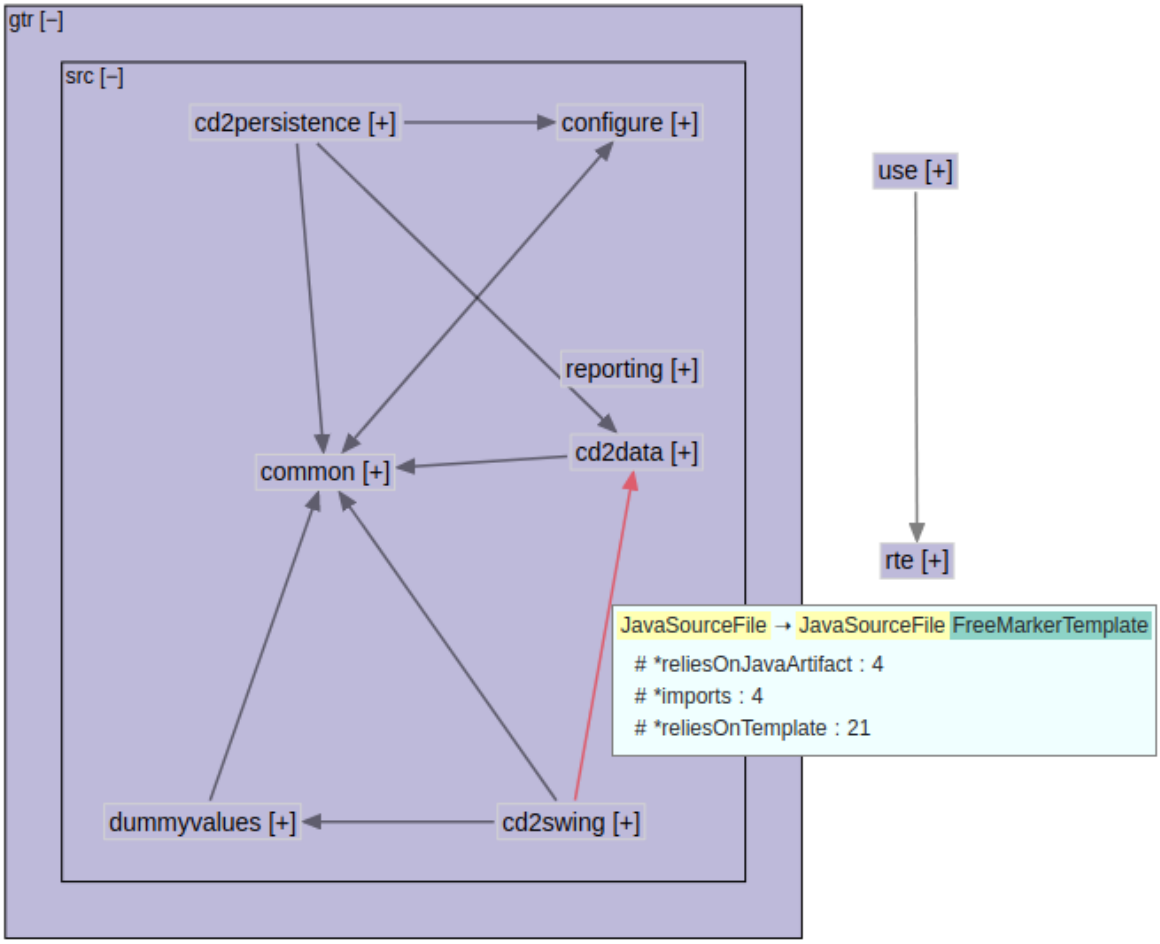

Abbildung 8.40.: Hierarchische Graphvisualisierung von Artefaktdaten mit ArtViz.

Weitere von ArtViz unterstützte Diagrammarten sind Balken-, Säulen- und Streudiagramme. Zu beachten ist, dass in Balken- und Säulendiagrammen zwei verschiedene Metriken und im Streudiagramm sogar drei verschiedene Metriken zusammen dargestellt werden können. Wie in Abbildung 8.43 gezeigt, stellt das Balkendiagramm dabei die Maßzahlen der Metriken nebeneinander dar. Demgegenüber bildet das Streudiagramm je eine Metrik auf der x- und y-Achse und die dritte Metrik durch die Größe der Punkte ab.

Welche Metriken zusammen in welcher Tabelle dargestellt werden und welche Diagrammarten jeweils pro Metrik zur Auswahl stehen, kann durch den *Administrator* festgelegt werden (siehe Abschnitt 9.4).

Afferente Kopplung, Eff, Ins | LoC (Template), Ins, Gen, Inc, Inc | Contains | Gets Generated | References, Ref

Metrics: Afferente Kopplung,Efferente Kopplung,Instabilitaet

Types: JavaClass,JavaEnum,JavaInterface

Artifacts	Type	Afferente Kopplung	Efferente Kopplung	Instabilitaet
@A	JavaClass	2	0	0
@PersistenceStorageTrafc	JavaClass	0	73	1
@B	JavaClass	1	1	0.5
@AbstractObservable	JavaClass	1	8	0.8889
@IPresenter	JavaInterface	5	0	0
@C	JavaClass	0	2	1
@ImageResources	JavaInterface	39	0	0
@StatusBar	JavaClass	12	4	0.25
@SouthPanePresenter	JavaClass	2	35	0.9459
@TableModelFactory	JavaClass	0	0	0
@Response	JavaClass	22	0	0
@TypeTableModelsTrafo	JavaClass	0	48	1
@ConcreteListPanelPresei	JavaClass	0	70	1
@UpdateRequest	JavaInterface	1	0	0
@MainWindowContent	JavaClass	15	33	0.6875
@SouthPaneModel	JavaClass	0	5	1
@StatusBarWriter	JavaClass	3	4	0.5714
@ObserverTrafo	JavaClass	0	37	1
@DirtyStateChangeEventI	JavaInterface	9	2	0.1818
@ConcreteDummyServer/	JavaClass	4	2	0.3333

355 total 1 2 3 4 5

Abbildung 8.41.: Farbcodierung zur Darstellung normierter Maßzahlen mit MontiArT.

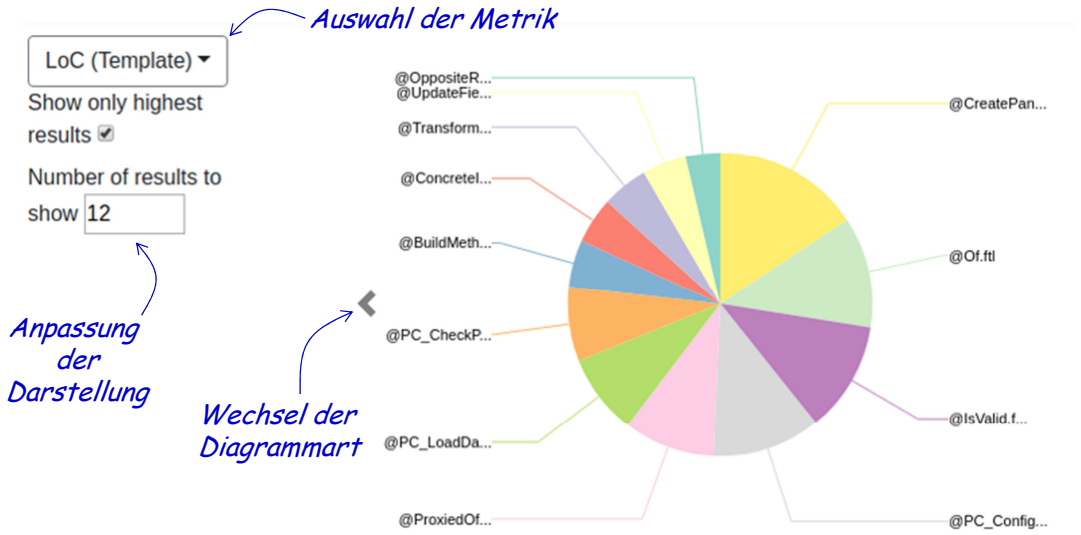

Abbildung 8.42.: Kuchendiagramm zur Darstellung von Metrikmaßzahlen mit ArtViz.

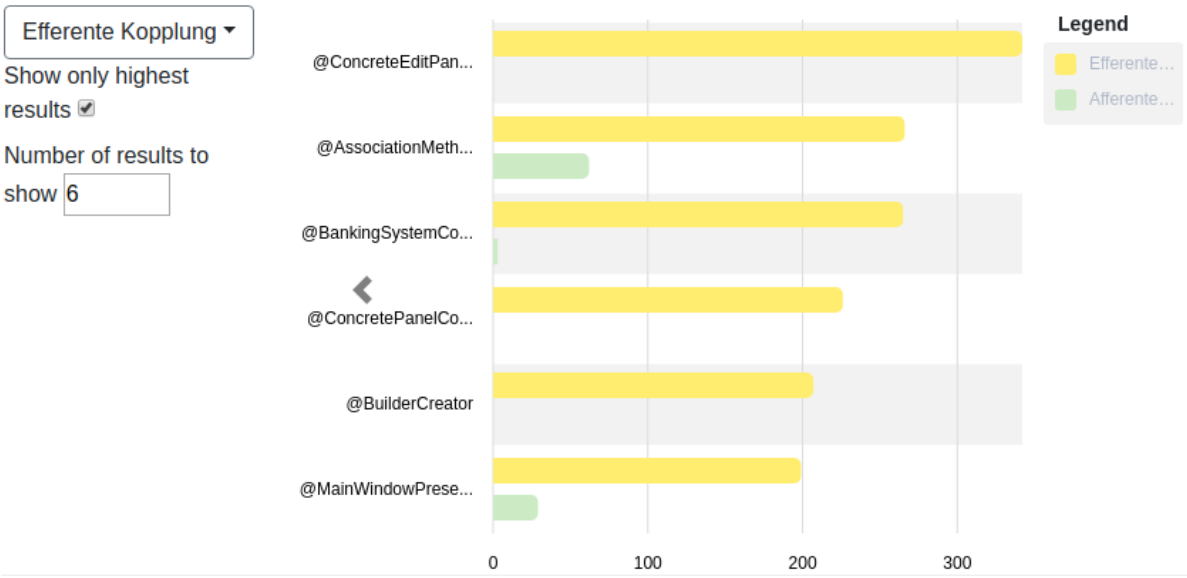

Abbildung 8.43.: Balkendiagramm zur Darstellung von Metrikmaßzahlen mit ArtViz.

8.6. Konverterwerkzeuge

In diesem Abschnitt werden kleinere Hilfswerkzeuge zur Konvertierung von Ein- und Ausgabeartefakten von MontiArT-Werkzeugen vorgestellt. Diese Werkzeuge führen keinen der in Kapitel 7 beschriebenen Schritte der artefaktbasierter Analysen direkt durch, sondern helfen bei der für den Zusammenschluss der Werkzeuge notwendigen Konvertierung von auszutauschenden Artefakten. Ihre Aufgaben sind die Entfernung hierarchischer OD-Anteile (Abschnitt 8.6.1), die Überführung von Artefaktdaten in ein für die Visualisierung benötigtes JSON-Format (Abschnitt 8.6.2), die Aufteilung der Informationen auf verschiedene Artefakte (Abschnitt 8.6.3) sowie das Mergen von Klassendiagrammanteilen (Abschnitt 8.6.4).

Der `ODMerger` würde ebenfalls gut diese Kategorie von Werkzeugen passen, wurde aber wegen seiner zentralen Rolle beim Mergen bereits in (Abschnitt 8.3.1) beschrieben.

8.6.1. OD Flattener

Der `ODFlattener` ist ein Werkzeug, das die hierarchischen Anteile eines Eingabe-ODs entfernt und das so entstehende, flache OD als Ausgabe produziert. Dieser Arbeitsschritt wird im Folgenden auch *Flattening* genannt.

Ähnlich wie der `ODFilter` (vgl. Abschnitt 8.5.3) würde sich der `ODFlattener` ebenfalls gut in die `language-tools` einfügen, da er das Flattening unabhängig vom verwendeten AM durchführt. Da der `ODFlattener` ebenfalls bereits auf der aktuellen Version der OD-Sprache arbeitet, ist dies aktuell jedoch nicht möglich. Nach einer Migration der `language-tools` auf die neue OD-Version kann eine Integration stattfinden. Der `ODFlattener` wird, wie in Listing 8.44 dargestellt, aufgerufen.

```
1 java -jar od-flattener [Option] Hierarchical.od Flat.od
```

Listing 8.44: Schematischer Aufruf des `ODFlatteners`.

Zusätzlich zu der Option zum Anzeigen des Hilfetextes ist die in Tabelle 8.45 dargestellte Option verfügbar. Ein Report wird bei Ausführung des Werkzeugs nicht erstellt.

Kurz	Lang	Beschreibung
`-g`	`--generate`	Generiert Identifikatoren für anonyme Objekte im hierarchischen Anteil des Eingabe-ODs. Ohne diese Option wird der Arbeitsschritt bei Verarbeitung eines solchen Objekts mit einem *error* abgebrochen.

Tabelle 8.45.: Mögliche Option bei dem Aufruf des `ODFlatteners`.

Der vom `ODFlattener` durchgeführte *Flatteninglgorithmus* besteht aus den folgenden Schritten. Dabei wird das Eingabe-OD OD_{in} in das Ausgabe-OD OD_{out} transformiert.

1. Kopieren der Top-Level-Objekte von OD_{in} nach OD_{out}.

2. Kopieren der normalen (nichthierarchischen) Links von OD_{in} nach OD_{out}.
3. Transformieren von inneren Links aus OD_{in} zu normalen Links in OD_{out}.
4. Transformieren von hierarchisch verschachtelten Objekten aus OD_{in} zu Top-Level-Objekten in OD_{out}.

Die ersten beiden Schritte sind einfache Kopiervorgänge, bei denen die nichthierarchischen Anteile direkt übernommen werden. Anschließend wird der AST von OD_{in} traversiert. Für jeden inneren Link wird in Schritt (3) ein normaler Link mit gleichen Start- und Zielreferenzen in OD_{out} erstellt. Zusätzlich werden alle durch innere Links verbundenen Objekte zwischengespeichert. Da alle hierarchisch verschachtelten Objekte über innere Links verbunden sind, erhält man so eine Menge, in der aller hierarchisch verschachtelten Objekte vorhanden sind. Entfernt man aus dieser Menge die Top-Level-Objekte aus OD_{in}, erhält man die Menge der hierarchisch verschachtelten Objekte, die in Schritt (4) als Top-Level-Objekte zu OD_{out} hinzugefügt werden. Somit besteht OD_{out} nur aus Top-Level-Objekten und normalen Links, enthält aber denselben Informationsgehalt wie OD_{in}. Lediglich die hierarchischen Anteile von OD_{in} wurden entfernt.

Im Kontext von MontiArT haben alle Objekte der Artefaktdaten einen Identifikator. Nichtsdestotrotz erlaubt es die OD-Sprache auch anonyme Objekte zu definieren. Anonyme Objekte haben keinen Identifikator und können deshalb nur innerhalb von Hierarchien durch innere Links verknüpft werden. Das Flattening von anonymen Objekten über den oben beschriebenen Algorithmus ist nicht möglich, da die in Schritt (3) hinzugefügten Links Objekte anhand von deren Identifikatoren referenzieren. Durch die vorherige Generierung von Identifikatoren lassen sich auch ODs mit anonymen Objekten flatten. Abhängig von den übergebenen Optionen des `ODFlatteners` wird bei Auffinden von anonymen, hierarchischen Objekten der Arbeitsschritt mit Ausgabe einer Fehlermeldung abgebrochen oder ein Identifikator nach dem in Abschnitt 7.3 beschriebenen Algorithmus generiert.

In MontiArT wird der `ODFlattener` dazu verwendet, die Abwärtskompatibilität zwischen verschieden Sprachversionen der OD-Sprache zu erreichen. Während die mitgelieferten Extraktoren hierarchische ODs ausgeben, erwartet der `ODMerger` ODs der älteren Sprachversion als Eingabe (vgl. Abschnitt 8.3.1). Eine solche Transformation könnte auch als vorverarbeitender Schritt für MDD-Werkzeuge außerhalb von MontiArT nützlich sein, da diese dadurch weniger verschiedene Arten von Modellelementen verarbeiten können müssen, wodurch die Implementierung vereinfacht wird.

8.6.2. OD2JSON Transformation

Damit Artefaktdaten mit ArtViz (siehe Abschnitt 8.5.5) visualisiert werden können, müssen sie in ein JSON-Format [Int17b] überführt werden. MontiArT liefert zu diesem Zweck mit `OD2JSON` ein vom verwendeten AM unabhängiges Transformationswerkzeug mit. Als Eingabe wird ein OD übergeben und als Ausgabe eine JSON-Datei erzeugt. Dieses Werkzeug wird als Teil der `language-tools` ausgeliefert. Somit arbeitet es mit Version 0.2.2 der OD-Sprache.

Ein beispielhafter Aufruf von `OD2JSON` ist in Listing 8.46 dargestellt. Hier wird das OD `Data.od` als Eingabe verwendet und die Ausgabe in die Datei `Data.json` gespeichert. Neben der Hilfeoption (vgl. Abschnitt 8.1.2) sind keine weiteren Optionen verfügbar.

```
1 java language-tools.jar OD2JSON Data.od Data.json
```

Listing 8.46: Beispielhafter Aufruf des Werkzeugs `OD2JSON`.

`OD2JSON` parst zunächst das Eingabe-OD und transformiert den AST in eine eigene Datenstruktur. Aus dieser Datenstruktur kann mit Hilfe der Bibliothek Gson [GG118] direkt ein JSON-String berechnet werden. Dieser wird abschließend in die entsprechende Ausgabedatei geschrieben.

In MontiArT wird `OD2JSON` vor allem zur Vorbereitung für die Visualisierung benötigt. Außerhalb von MontiArT lässt sich das Werkzeug beispielsweise dann nutzen, wenn JSON statt OD als Austauschformat gewünscht oder benötigt wird.

8.6.3. CD OCL Splitter

Bei dem `CDOCLSplitter` handelt es sich um ein einfaches Werkzeug, welches die CD und OCL Anteile eines kombiniertes CD-Artefakts in zwei separate Artefakte aufteilt.

Im Gegensatz zu den bis hierhin vorgestellten Werkzeugen wurde der `CDOCLSplitter` durch ein *Shell-Skript* realisiert. Diese Möglichkeit wurde wegen der geringen Komplexität der Aufgabe gewählt. Somit dient dieses Werkzeug auch als Beispiel dafür, dass skriptbasierte Werkzeuge bei der Verwendung von MontiArT genutzt werden können. Listing 8.47 zeigt einen beispielhaften Aufruf des Werkzeugs.

```
1 sh CDOCLSplitter.sh Input.cd Output.cd Output.ocl
```

Listing 8.47: Beispielhafter Aufruf des `CDOCLSplitters`.

Ähnlich zu den anderen Werkzeugen werden bei Aufruf des `CDOCLSplitters` Eingabe- und Ausgabeartefakte als Parameter übergeben. In der aktuellen Version des Werkzeugs sind, anders als bei den anderen Werkzeugen, keinerlei Optionen – auch nicht die Standardoptionen für Werkzeuge (vgl. Abschnitt 8.1.2) – verfügbar. Die Behandlung der Hilfeoption lässt sich jedoch auch in einem skriptbasierten Werkzeug realisieren. Um zu den anderen Werkzeugen konforme Exitcodes zu erstellen, lässt sich der Befehl `exit` verwenden. Sowohl die Implementierung der Hilfeoption als auch die Rückgabe von Exitcodes sind für eine zukünftige Implementierung vorgesehen.

Listing 8.48 zeigt die skriptbasierte Implementierung des `CDOCLSplitters`. Das Skript ist in zwei Abschnitte aufgeteilt. Im ersten Abschnitt wird aus der Eingabe der CD-Anteil, im zweiten Abschnitt der OCL-Anteil extrahiert und ausgegeben. Mit Hilfe von `cat` [TOG18] wird jeweils der Inhalt des Eingabemodells an ein SED-Programm [TOG18] übergeben und

```sh
#!/bin/bash

#extract CD part
cat $1 |
sed -e '/context/,/;/ d' \
    -e '/inv:/,/;/ d' \
    -e "s/classdiagram.*{/classdiagram $(basename $2 .cd) {/" |
cat -s > $2

#extract OCL part
cat $1 |
sed -n '/context\|inv:/,/;/ p' |
sed -e "1 i\OCL $(basename $3 .cd) {\n\n" \
    -e '/;/G' \
    -e '$a\}' |
cat -s > $3
```

Listing 8.48: Implementierung des `CDOCLSplitters`.

am Ende in die jeweilige Ausgabedatei geschrieben. Bei der Ausgabe werden durch Angabe der Option `-s` doppelte Leerzeilen entfernt. Mit SED wird im Eingabemodell nach OCL-Ausdrücken gesucht. Im ersten Abschnitt werden die gefundenen OCL-Ausdrücke einfach entfernt, während im zweiten Abschnitt ausschließlich die gefunden OCL-Ausdrücke übernommen werden. Zusätzlich werden die Namen der entstehenden Modelle den Parametern entsprechend angepasst.

Zum Zeitpunkt der Entstehung dieser Arbeit war es durch die bestehenden Werkzeuge nicht möglich, beide Anteile in einem einzelnen Artefakt zu verarbeiten. Mit Hilfe des hier vorgestellten Werkzeugs können die Anteile in einzelne Artefakte aufgeteilt werden, die separat als Eingaben für andere Werkzeuge (siehe Abschnitte 8.4.2, 8.4.3, 8.5.1 und 8.5.2) dienen. Da das Werkzeug in der aktuellen Form als Skript realisiert ist, ist eine programmatische Verwendung in einem Java Programm nicht möglich und somit Punkt *TRQ-3.3* für dieses Werkzeug ausnahmsweise nicht erfüllt.

8.6.4. CD Merger

Neben den in im Rahmen dieser Arbeit entwickelten und bereits vorgestellten Werkzeugen, wird in MontiArT mit dem `CDMerger` ein weiteres Werkzeug verwendet. Dieses Werkzeug wird ebenfalls am Lehrstuhl für Software Engineering entwickelt und dient der Zusammenführung von CDs.

Der `CDMerger` wird in einem ausführbaren Archiv ausgeliefert. Da er jedoch nicht als Teil von MontiArT entwickelt wurde, werden die in Abschnitt 8.1 dargestellten Eigenschaften und Konventionen an dieser Stelle nicht eingehalten. Der `CDMerger` bekommt zwei CDs als

Eingabe und führt sie zu einem einzigen CD zusammen. Optionen können bei dem Aufruf des Werkzeugs nicht mit angegeben werden.

Bei der Ausführung des `CDMergers` werden gleichnamige Klassen des CDs verschmolzen. Dabei werden zugehörige Attribute nur zusammengefügt wenn sie kompatibel sind, d.h., Attribute müssen den gleichen Namen haben und ihre Datentypen müssen kompatibel sein (z.B. `long` und `int`). Assoziationen können zusammengeführt werden, wenn sie den gleichen Namen und die gleichen Start- und Zielklassen haben. Dabei müssen auch ihre jeweiligen Rollennamen und Kardinalitäten gleich oder für eine der zusammenzuführenden Assoziationen unterspezifiziert sein.

Durch die Auslieferung des Werkzeugs als ausführbares Jar-Archiv lässt sich der `CDMerger` gut in MontiArT integrieren. Dabei kann das Werkzeug sowohl manuell über die Kommandozeile, als Teil einer Werkzeugkette oder auch programmatisch genutzt werden. In MontiArT ist der Einsatz des `CDMergers` an verschiedenen Stellen hilfreich. Zum einen können so projektspezifische AMe aus dem AM-Kern, Anteilen bestehender AMe und projektspezifischen Anteilen komfortabel erstellt werden und zum anderen lassen sich so Erweiterungen zu Analysezwecken (vgl. Abschnitt 7.2.1) mit eine AM zusammenführen.

Kapitel 9.

Methodik zur Erstellung von MontiArT-Instanzen

Während in Kapitel 8 die Funktionsweise von MontiArT beschrieben wurde, wird in diesem Kapitel eine Methodik zur Erstellung von MontiArT-Instanzen vorgestellt, was die Verwendung, die Konfiguration und die Anpassung der Werkzeugkette mit einschließt. Eine MontiArT-Instanz ist dabei als projektspezifische, konfigurierte und angepasste Instanz der Werkzeugkette MontiArT definiert (siehe Definition 8.0.1).

Abbildung 9.1 gibt eine Übersicht über die notwendigen Schritte zur Erstellung einer MontiArT-Instanz und zeigt die Abschnitte dieses Kapitels, in denen die einzelnen Schritte detailliert werden.

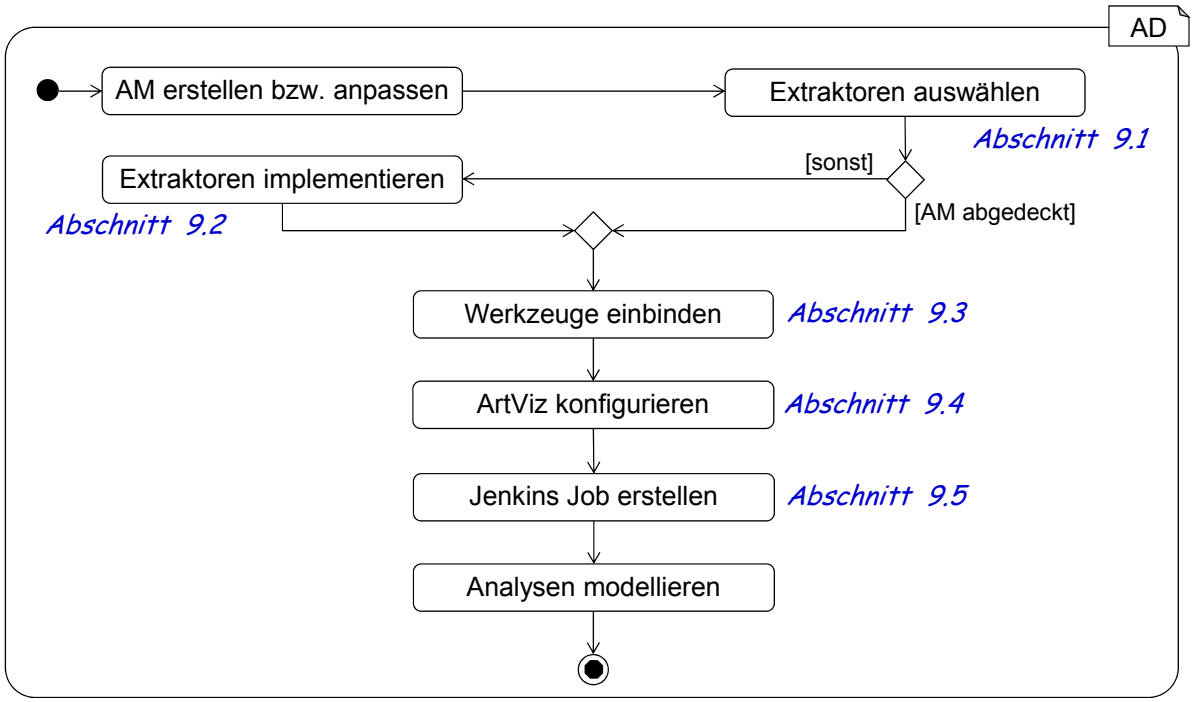

Abbildung 9.1.: Methodik zur Verwendung der MontiArt-Werkzeugkette.

Bei dem dargestellten Ablauf wird davon ausgegangen, dass sich die Anreicherung vollständig OCL-basiert durchführen lässt und die Analysen vollständig modelliert und von dem `OCL-Generator` verarbeitet werden können. Dadurch ist es möglich, Anpassungen der Analysen unabhängig von der Konfiguration der MontiArT-Instanz durchzuführen, sodass zur Anpassung einer Analyse lediglich eine Anpassung der zugehörigen modellbasierten Spezifikationen nötig ist. In Kapitel 8 wurde die Verwendung von projektspezifischen Analysewerkzeugen zur Durchführung des Analyseschritts als Alternative zu OCL-basierten Analysen angegeben. Wird auf die Verwendung solcher Werkzeuge zurückgegriffen, müssen unter Umständen bei Änderung der Analysen entsprechende Werkzeuge angepasst oder sogar neue Werkzeuge hinzugefügt werden. Letzteres führt dabei zwingend zu einer Konfigurationsanpassung der MontiArT-Instanz.

Während für den ersten Schritt *AM erstellen bzw. anpassen* bereits eine eigene Methodik in Abschnitt 7.1 und für den letzten Schritt *Analysen modellieren* eine Methodik in Abschnitt 7.2 beschrieben wurde, wird in den folgenden Abschnitten auf die restlichen Schritte eingegangen. Dabei ist zu beachten, dass zwar eine vollständig modellbasierte Analyse angestrebt wird, darüber hinaus aber durch die gewählte Architektur von MontiArT die Möglichkeit zur komfortablen Erstellung und zum Einbinden eigener Werkzeuge gegeben ist. Abschnitt 9.2 beschränkt sich aus diesem Grund nicht ausschließlich auf die Erstellung von Extraktoren, sondern befasst sich allgemeiner mit der Implementierung projektspezifischer MontiArT-Werkzeuge.

9.1. Auswahl mitgelieferter Extraktoren

Die Auswahl von mitgelieferten Extraktoren ist ein kleiner Schritt bei der Erstellung von MontiArT-Instanzen. Hier muss geprüft werden, ob das aus dem Schritt *AM erstellen bzw. anpassen* hervorgehende Modell vollständig durch die vorhandenen Extraktoren abgedeckt werden kann. Solche Extraktoren können entweder durch MontiArT mitgelieferte oder projektspezifische Extraktoren aus anderen Projekten sein, die sich im Kontext des aktuellen Projekts wiederverwenden lassen. Das AM ist bezüglich der angestrebten Analysen abgedeckt, wenn sich alle für diese Analysen relevante Daten erheben lassen. Es existiert dann für alle relevanten Klassen, deren relevante, nichtabgeleitete Attribute aus AM_{CD} sowie die relevanten Assoziationen ein Extraktor, der die dazugehörigen Daten extrahiert. Relevante abgeleitete Eigenschaften können ebenfalls extrahiert oder in dem späteren Schritt der Anreicherung (vgl. Abschnitt 7.5.2) berechnet werden. Unter der Annahme, dass die Berechnungsvorschrift durch entsprechende OCL-Ausdruck modelliert wurde, lässt sich das Werkzeug zur Anreicherung vollständig generieren (siehe Abschnitt 8.4.5).

9.2. Implementierung projektspezifischer Werkzeuge

In diesem Abschnitt wird auf die Implementierung projektspezifischer Werkzeuge eingegangen. Dies umfasst die Implementierung neuer Werkzeuge unter Verwendung von wieder-

verwendbaren Klassen (Abschnitt 9.2.1), die Extraktion von Daten mittels Symboltabelle (Abschnitt 9.2.2) sowie die Verwendung und Anpassung der Reporting Infrastruktur von MontiCore (Abschnitt 9.2.3). Die letzten beiden Punkte sind MontiCore-spezifisch und definieren ein systematisches Vorgehen zur Extraktion bestimmter Informationsarten.

9.2.1. Implementierung neuer Werkzeuge

Bei der Implementierung neuer Werkzeuge kann das gleiche Vorgehen genutzt werden, das auch schon bei der Implementierung der mitgelieferten Werkzeuge angewandt wurde. Die in Abbildung 8.5 und Abbildung 8.10 bereits vorgestellten Klassen `AToolCli` und `AExtractor` können und sollten zur Implementierung projektspezifischer Werkzeuge wiederverwendet werden.

Durch die Verwendung der Oberklassen wird die Implementierung von Werkzeugen vereinheitlicht. Dies bietet den Vorteil, dass sich Werkzeuge einfacher von *Analysten* nutzen lassen, da sie sich analog zu den mitgelieferten Werkzeugen aufrufen lassen. Außerdem wird die Wartung und Weiterentwicklung der Werkzeuge durch *Analysewerkzeugentwickler* durch die gleichen internen Strukturen der verschiedenen Analysewerkzeuge vereinfacht.

Darüber hinaus bieten diese Klassen Funktionalität, die die Entwicklung neuer Werkzeugen erleichtern. Die Klasse `AToolCLI` stellt Funktionalität für die Verarbeitung von Parametern bei Aufruf von Werkzeugen eines ausführbaren Jar-Archivs, die Behandlung der verschiedenen Zustände, die Ausgabe von Nachrichten und die Rückgabe der Exitcodes bereit. Die eigentliche Werkzeugklasse kann dann in der Methode `execute` der konkreten CLI-Klasse aufgerufen werden (vgl. Abschnitt 8.1.3). Für statische Extraktoren kann zusätzlich die abstrakte Klasse `AExtractor` verwendet werden, welche Funktionalität zur Behandlung von `rootDirs` bietet.

Im Gegensatz zur Implementierung statischer Extraktoren hängt die Implementierung dynamischer Extraktoren stark von der Techologie des zu untersuchenden Projektteils ab. Für MontiCore wird in Abschnitt 9.2.3 eine Methodik zur Implementierung dynamischer Extraktoren basierend auf der Reportinginfrastruktur vorgestellt.

9.2.2. Extraktion von Artefaktbeziehungen mit Hilfe der Symboltabelle

Zur Extraktion von Beziehungen zwischen Modellartefakten MontiCore-basierter Sprachen können vorhandene Symboltabellenimplementierungen genutzt werden. Ist die Symboltabelle zu einem Modellartefakt erstellt, kann ihr *Resolvemechanismus* zur Auflösung von Referenzen zu Elementen anderer Artefakte genutzt werden. Ein Quellartefakt steht in Beziehung zu einem anderen Artefakt, wenn ein zu dem Quellartefakt zugehöriges Referenzsymbol ein Symbol des anderen Artefakts referenziert. Dies kann sowohl zwischen Artefakten derselben Art als auch zwischen Artefakten verschiedener Arten der Fall sein. Für Modelle gilt letzteres bei der Verwendung von Sprachaggregation (vgl. Abschnitt 2.1.1).

Für MontiCore-basierte Werkzeuge, deren verwendete Sprachen eine Symboltabellenimplementierung bereitstellen, können die Beziehungen zwischen den Modellartefakten durch eine systematische Vorgehensweise extrahiert werden. Dabei wird vorausgesetzt, dass die Implementierung der Symboltabelle, der in [MSN17] beschriebenen Systematik folgt. Abbildung 9.2 zeigt die Methodik zur Extraktion von Beziehungen zwischen Artefakten mit Hilfe der Symboltabelle.

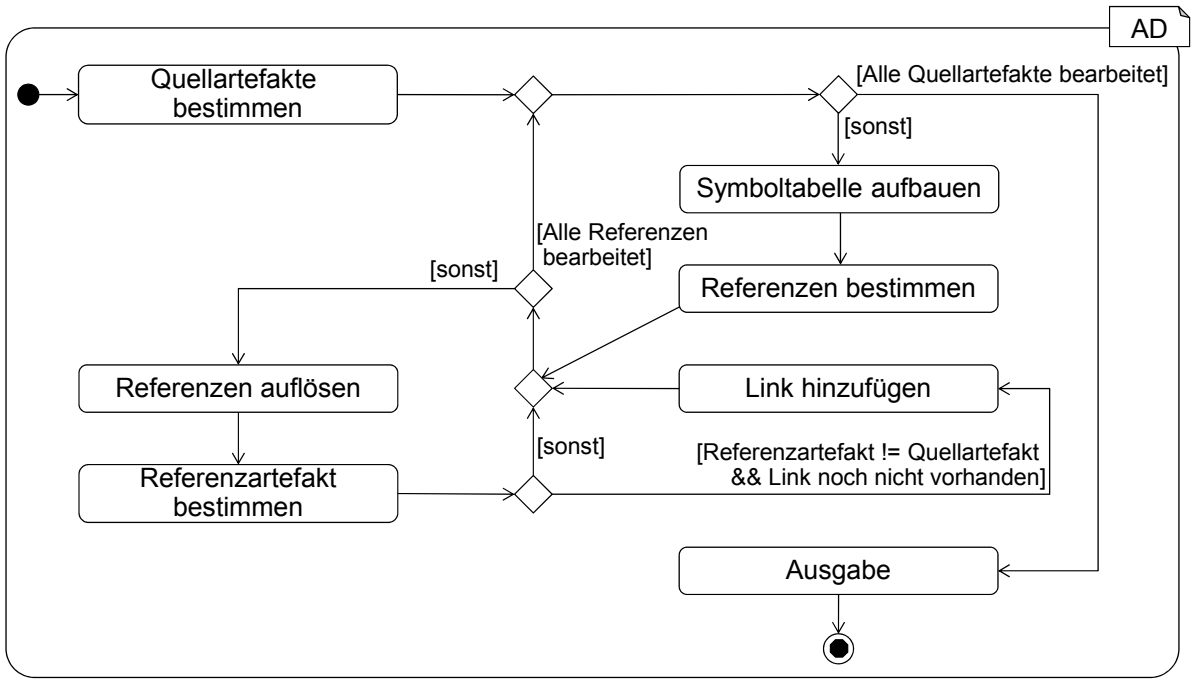

Abbildung 9.2.: Methodik zur Extraktion von Beziehungen zwischen Artefakten mit Hilfe der Symboltabelle.

Zunächst werden alle relevanten Artefakte bestimmt, für die Beziehungen zu anderen Artefakten extrahiert werden sollen. Für jedes dieser Artefakte, werden in den folgenden Schritten die ausgehenden Beziehungen berechnet, weshalb sie in Abbildung 9.2 als Quellartefakte bezeichnet werden.

Für jedes Quellartefakt wird bei seiner Verarbeitung zunächst der entsprechende Teil der Symboltabelle aufgebaut. Um diesen Schritt durchzuführen, kann entweder die sprachspezifische Subklasse des `SymbolTableCreators` der Symboltabelleninfrastruktur [MSN17] direkt genutzt werden oder alternativ das Top-Level-Symbol unterhalb des `ArtifactScopes` geladen werden. In beiden Fällen führt dies dazu, dass die Symboltabelle für das Quellartefakt aufgebaut und mit dem AST verknüpft wird. Zu beachten ist hier, dass die Struktur der Symboltabelle sowie die Verknüpfungen zwischen Symboltabelle und dem AST sprachspezifisch festgelegt werden. Nach diesem Schritt wird davon ausgegangen, dass sowohl der AST als auch die Symboltabelle zur Verfügung stehen.

Im nächsten Schritt werden alle Referenzen bestimmt. Referenzen werden üblicherweise im AST durch spezielle Referenzknoten und in der Symboltabelle über spezielle Referenzsymbole repräsentiert. Die entsprechenden Objekte der Datenstrukturen lassen sich somit anhand ihres Typs als Referenz erkennen.

Abhängig von der Struktur der Symboltabelle und der entsprechenden Verknüpfung mit dem AST gibt es mehrere Möglichkeiten die Referenzen des Quellartefakts zu berechnen. Zum einen können die entsprechenden Stellen im AST gesucht werden. Existieren entsprechende Verknüpfungen mit der Symboltabelle, kann hier auch direkt zu den verknüpften Symbolen navigiert werden, was das Auflösen der Referenzen im nächsten Schritt vereinfacht. Die Suche im AST hat den Vorteil, dass die von MontiCore bereitgestellte Implementierung des Visitorentwurfsmusters (vgl. Abschnitt 2.3.1) genutzt werden kann, wodurch sich die entsprechende Suche sehr einfach implementieren lässt.

Alternativ dazu lässt sich die Symboltabelle direkt traversieren. Sind alle notwendigen Informationen in der Symboltabelle vorhanden, können durch eine Suche im `Scopebaum` der Symboltabelle alle Symbole besucht und Referenzsymbole herausgefiltert werden.

Für welche Variante sich ein *Analysewerkzeugentwickler* an dieser Stelle entscheidet, hängt von der Implementierung der jeweiligen Symboltabelle, aber auch von der Analyseart ab. Sollen beispielsweise bei der Auflösung desselben Referenzsymboltyps verschiedene Referenzarten extrahiert werden, so muss neben dem Typ des Symbols auch die Stelle innerhalb der Datenstruktur betrachtet werden.

Nach Bestimmung der Referenzen für das aktuelle Quellartefakt müssen sie im nächsten Schritt aufgelöst werden. Dabei wird jede Referenz mit Hilfe der Symboltabelle einzeln anhand des Namens des referenzierten Symbols aufgelöst. Dazu kann, bei ausschließlicher Verwendung qualifizierter Namen, der Vorgang von einem beliebigen `Scope` der Symboltabelle gestartet werden. Bei Verwendung von einfachen Namen (zum Beispiel in Kombination mit Import Statements) muss der entsprechende `EnclosingScope` als Start des Vorgangs genutzt werden. Der Vorgang lässt sich mit der `resolve` Methode des in Abbildung 9.3 gezeigten `Scope`-Interfaces starten.

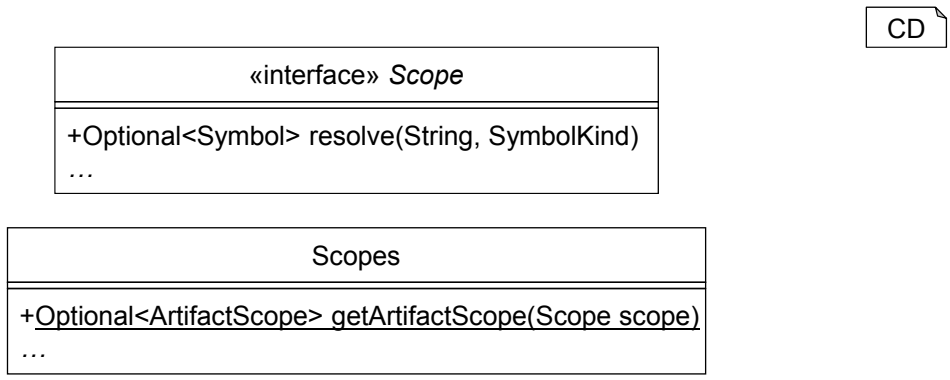

Abbildung 9.3.: Das `Scope`-Interface und die `Scopes`-Klasse.

Wichtig ist hier, dass der `resolve`-Methode der Typ des aufzulösenden Symbols (`SymbolKind`) als Parameter übergeben werden muss. In den Fällen, in denen der Referenzsymboltyp nicht eindeutig ist, muss ein Vorgang für jeden möglichen Typ durchgeführt werden. Ein Beispiel für einen solchen Fall sind Typen (siehe Abschnitt 6.1.1), welche sowohl durch Java- als auch durch CD-Artefakte definiert werden können. Somit steht nun das referenzierte Symbol aus der Symboltabelle des referenzierten Artefakts für den nächsten Schritt zur Verfügung.

Als nächster Schritt muss noch das zu dem Symbol zugehörige Artefakt bestimmt werden. Dazu kann die Methode `getArtifactScope` der `Scopes`-Klasse (ebenfalls Abbildung 9.3) genutzt werden. Mit Hilfe der in dem `ArtifactScope` beinhalteten Informationen lässt sich leicht das zugehörige Artefakt bestimmen. Unterscheidet sich dieses Referenzartefakt von dem Quellartefakt, wird ein entsprechender Link in den Artefaktdaten erstellt.

Ein Beispiel für die Anwendung der Methodik findet sich später in Abschnitt 10.2. Dort werden die Beziehungen zwischen Modellen einer Sprachfamilie bestehend aus MAA-Modellen, CDs und Java-Dateien untersucht. Dies ermöglicht den *Modellierern* die Übersicht über die Verknüpfungen auf Modellebene zu behalten sowie bei anstehenden Änderungen eine Change-Impact-Analyse (Abschnitt 3.4.2) auf Modellebene durchzuführen.

Ein Werkzeug, welches unabhängig von den verwendeten Artefaktarten und basierend auf der Symboltabelle die Beziehungen zwischen Artefakten extrahiert, ist aktuell nicht realisierbar. Dies ist durch die oben erwähnten, möglichen sprachspezifischen Unterschiede der Symboltabellenimplementierungen begründet.

9.2.3. Verwendung und Anpassung der Reportinginfrastruktur

Um dynamische Beziehungen zur Laufzeit von MontiCore-Werkzeugen zu extrahieren, bietet sich die in Abschnitt 2.3.4 vorgestellte Reportinginfrastruktur an. In diesem Abschnitt wird die Methodik zur Verwendung und Anpassung der Reportinginfrastruktur beschrieben.

Zunächst muss entschieden werden, ob die Reportinginfrastruktur direkt verwendet werden kann oder ob Anpassungen notwendig sind. Dies hängt maßgeblich von den zu extrahierenden Informationen und der bereits vorhandenen Nutzung der Reportinginfrastruktur im zu untersuchenden Werkzeug ab. Abbildung 9.4 dient hierbei als Entscheidungshilfe.

Lassen sich die Informationen aus den Events ableiten, die durch das Werkzeug bereits reportet werden, so ist keine weitere Instrumentierung des Werkzeugs nötig. Stattdessen kann durch den *Analysewerkzeugentwickler* ein zusätzlicher Reporter geschrieben werden, welcher als Observer [GHJV95] für Reportingevents registriert wird und entsprechend auf solche Events reagiert. Ein solcher Reporter muss das Interface `IReportEventHandler`, welches Teil der MontiCore-RTE ist, implementieren. Ein Beispiel für einen projektspezifischen Reporter lässt sich in Abschnitt 10.1.2 finden.

Sollten die entsprechenden Events noch nicht durch das Werkzeug reportet werden, so ist zusätzlich eine Instrumentierung des Werkzeugs notwendig. An den Stellen im Werkzeug, die zur Auslösung eines Reportingevents führen, muss die entsprechende statische Methode der

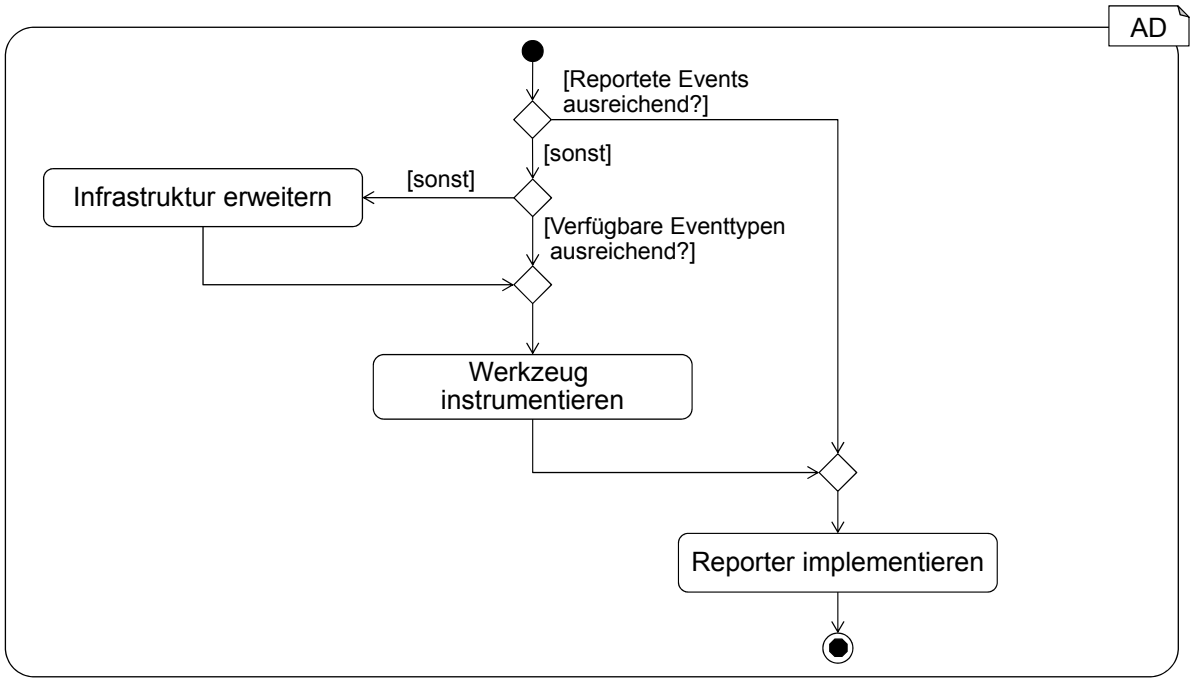

Abbildung 9.4.: Entscheidungshilfe für die Verwendung und Anpassung der Reportinginfrastruktur.

`Reporting`-Klasse (siehe [HR17]) aufgerufen werden. Anschließend kann wie im vorherigen Fall ein eigener Reporter implementiert und registriert werden.

Der dritte denkbare Fall ist, dass die zur Verfügung stehenden Reportingevents nicht ausreichen. Das Hinzufügen neuer Reportingevents lässt sich unter ausschließlicher Verwendung der Reportinginfrastruktur nicht lokal durch Instrumentierung des zu untersuchenden Werkzeugs beheben. An dieser Stelle gibt es zwei Möglichkeiten, um trotzdem sowohl auf bestehende Reportingevents als auch auf die neu hinzuzufügenden Events reagieren zu können. Die erste Möglichkeit ist die Anpassung der Reportinginfrastruktur im Kern von MontiCore. Dies sollte vor allem dann geschehen, wenn abzusehen ist, dass Events dieser Art für eine Vielzahl von MontiCore-basierten Werkzeugen von Interesse sind. Langfristig kann auch über eine flexiblere Handhabung von Reportingevents innerhalb der Reportinginfrastruktur nachgedacht werden, um das Hinzufügen neuer Events auch ohne Anpassungen im Kern der Infrastruktur zu ermöglichen. Die zweite Möglichkeit ist, dass ein zusätzlicher Mechanismus zum Reporten von eigenen Events in das Werkzeug integriert wird. Dabei muss darauf geachtet werden, dass der Mechanismus dem neu zu entwickelnden Reporter erlaubt diese Events zusätzlich zu denen der Reportinginfrastruktur zu empfangen.

Eine Alternative zur Verwendung der Reportinginfrastruktur lässt sich mit Hilfe des Monitoringframeworks Kieker [vHWH12, vHRH+09] realisieren. In [Con16] wurde gezeigt, das sich die Verwendung der Reportinginfrastruktur grundsätzlich durch eine Kieker-basierte Extraktion austauschen lässt.

9.3. Fehlererkennung bei skriptbasierter Verkettung von Werkzeugen

MontiArT-Werkzeuge lassen sich auf verschiedene Arten zu einer Werkzeugkette zusammenschließen. Zum einen kann ein neues Werkzeug geschrieben werden, welches alle anderen Werkzeuge als Unterwerkzeuge nutzt und somit auf die Programmierschnittstelle (engl. *Application Programming Interface* (API)) der einzelnen Werkzeuge zugreift. Alternativ lassen sich die Werkzeuge mit Hilfe eines Groovy-Skriptes verketten, sodass ebenfalls die APIs der Werkzeuge direkt genutzt werden können, jedoch kein zusätzlicher Übersetzungsschritt notwendig ist. Eine dritte Alternative ist die Verkettung per Skript, bei der die ausführbaren Jar-Archive der Werkzeuge genutzt werden. Dieser dritte Fall wird im Folgenden weiter betrachtet. Dabei werden Shell-Skripte verwendet. Ein solches Shell-Skript wird für diesen Zweck in einem später gezeigten Anwendungsfall (Abschnitt 10.1) vorgestellt. Bei Nutzung eines solchen Skriptes ist es wichtig, dass ein Fehlschlag eines Werkzeugs erkannt und die Ausführung der Werkzeugkette gestoppt wird, da ansonsten fehlerhafte Analyseergebnisse erstellt und unter Umständen weiterverarbeitet werden.

Wie in Abschnitt 8.1 beschrieben, geben Werkzeuge bei Auftritt eines Fehlers einen Fehlercode zurück. Damit das Skript entsprechend auf einen Fehler bei der Ausführung eines Werkzeugs reagiert, muss die entsprechende Fehlerbehandlung in dem Skript hinzugefügt werden. Dazu kann die in Listing 9.5 gezeigte Funktion zur Fehlerbehandlung genutzt werden.

```sh
PROGNAME=$(basename $0)
function error_exit
{
  echo "${PROGNAME} $1: Tool exits with error code $2" 1>&2
  exit 1
}
```

Listing 9.5: Abbruchfunktion bei Nutzung einer Shell-Skript-basierten MontiArT-Instanz.

Die Funktion erhält zwei Parameter, die Zeilennummer des Werkzeugaufrufs im Skript und den Fehlercode. Der Programmname lässt sich über `$(basename $0)` ermitteln. `$0` enthält dabei den relativen Pfad des Skriptes, der zum Aufruf verwendet wurde. Mit `basename` wird dann der Name des Skriptes selbst berechnet. Der Befehl `1>&2` schreibt schlussendlich den berechneten String auf die Standardfehlerausgabe. Die Abbruchfunktion kann wie in Listing 9.6 dargestellt aufgerufen werden.

```sh
java -jar ExecutableJarArchive.jar || error_exit $LINENO $?
```

Listing 9.6: Beispielhafter Aufruf der Abbruchfunktion des mitgelieferten Shell-Skripts.

Das dargestellte Schema kann immer dann verwendet werden, wenn ein Fehler bei der Ausführung eines Werkzeugs auch zum Abbruch der Ausführung der gesamten Werkzeugkette führen soll. Der Kontrolloperator || ist wie folgt definiert:
"An or list has the form
command1 || *command2*
command2 is executed if, and only if, command1 returns a non-zero exit status." [RF16]

Somit wird die Abbruchfunktion nur aufgerufen, wenn der Werkzeugaufruf einen Status ungleich `0` zurückliefert. Die für den Aufruf benötigten Parameter lassen sich automatisch über die Variablen `$LINENO` und `$?` ermitteln. Der Aufruf aus Listing 9.6 führt bei einem Rückgabewert von `1` zu der folgenden Fehlermeldung:

```
montiart.sh 9: Tool exits with error code 1
```

Zusätzlich produzieren die in dieser Arbeit beschriebenen Werkzeuge eigene Fehlermeldungen, die vor der Rückgabe des Fehlercodes auf die Standardfehlerausgabe geschrieben werden. Bei Verwendung der Analysewerkzeuge muss außerdem auf den Rückgabewert 2 gesondert reagiert werden, damit bei einer fehlgeschlagenen Prüfung die Ausführung der Werkzeugkette trotzdem fortgesetzt werden kann (vgl. Abschnitt 8.5.2). In diesen Fällen kann statt der oben vorgestellten Funktion die in Listing 9.7 dargestellte Funktion genutzt werden. Diese gibt lediglich eine Warnung aus wenn von einem Werkzeug ein *failure* (siehe Abschnitt 8.1.1) zurückgegeben wurde. In anderen Fällen wird die `error_exit`-Funktion aus Listing 9.5 aufgerufen. Von dort wird auch die Definition der Variablen `PROGNAME` übernommen.

SH
```
function warn_error_exit
{
  if [ $2 -eq 2 ]; then
    echo "${PROGNAME} $1: Warning - tool exits with error code $2"
  else
    error_exit $1 $2
  fi
}
```

Listing 9.7: Erweiterte Fehlerbehandlung, bei der die Ausführung der Werkzeugkette im *failure*-Fall unter Ausgabe einer Warnung fortgesetzt wird.

Das beschriebene Verfahren zur Behandlung von Fehlercodes orientiert sich an dem Tutorial [SJ17] zu dem Buch [SJ16]. Weitere Werkzeuge können unter Verwendung dieses Schemas einfach eingebunden werden.

Zu beachten ist, dass dieses Verfahren nur für Shell-Skripte eingesetzt werden kann. Bei der Verwendung von anderen Sprachen müssen entsprechend andere Lösungen gefunden werden. Für Make lässt sich beispielsweise der Ausdruck aus Listing 9.8 verwenden, um die Ausführung einer Regel trotz eines Rückgabewerts von 2 nicht abzubrechen.

```Make
target:
  java -jar ExecutableJarArchive.jar || [ $$? -eq 2 ] || \
  (echo "Error"; exit 1;)
```

Listing 9.8: Fehlerbehandlung mit Make.

Im Fehlerfall (bei einem Rückgabewert ungleich 0) wird geprüft, ob der Rückgabewert 2 ist. Ist dies nicht der Fall, wird die Fehlermeldung „Error“ ausgegeben und die Ausführung der Regel abgebrochen.

9.4. Konfiguration von ArtViz

Um Artefaktdaten mit ArtViz anzeigen zu können, müssen die Daten der Webseite zur Verfügung gestellt und ArtViz entsprechend konfiguriert werden.

Um die Daten der Webseite zur Verfügung zu stellen, muss zunächst ein Projekt angelegt werden. Dazu muss im `assets` Ordner der Webseite ein Unterordner erstellt werden.

Unterordner des `assets`-Ordners werden von ArtViz als Projekt Ordner (`PROJEKT_ORDNER`) interpretiert. Um ein neues Projekt anzulegen, muss an dieser Stelle lediglich ein weiterer `PROJEKT_ORDNER` angelegt werden. Die Ordnerstruktur des `DATA`-Ordners wird später in Listing 9.12 dargestellt.

Über eine JSON-Konfigurationsdatei lässt sich das `defaultProject` angeben, welches beim Aufrufen von ArtViz zu Beginn ausgewählt wird. Eine Alternative hierzu wäre, zunächst eine Liste aller Projekte anzuzeigen, aus der dann ein Projekt ausgewählt werden muss. Listing 9.9 zeigt beispielhaft die Konfiguration des `defaultProjects` mit Hilfe der Konfigurationsdatei. Diese Datei muss `globalProperties.json` genannt werden und in dem `assets` Ordner abgelegt werden.

```JSON
{
 "defaultProject" : "Dex-ArtifactAnalysis-Test"
}
```

Listing 9.9: Konfiguration des initial ausgewählten Projekts mit Hilfe der `globalProperties.json`-Datei.

Bei einem leeren `PROJEKT_ORDNER` zeigt ArtViz bereits das zugehörige Projekt in der *textuellen Ansicht* (vgl. Abschnitt 8.5.5) ohne Daten an. Die Artefaktdaten im OD-Format müssen in einem Unterordner mit dem Namen `textFiles` innerhalb des `PROJEKT_ORDNERS` abgelegt werden. Dabei sind beliebige Unterordnerstrukturen möglich, die dann entsprechend in der textuellen Ansicht von ArtViz dargestellt werden. Damit Daten auch als Graph in

der *Graphansicht* dargestellt werden können, müssen sie zusätzlich im JSON-Format (siehe Abschnitt 8.6.2) neben den entsprechenden OD-Dateien vorliegen.

Um die hierarchische Graphansicht sowie die Metrikansicht nutzen zu können, bedarf es weiterer projektspezifischer Konfigurationen.

Listing 9.10 zeigt den Teil der projektspezifischen Konfigurationsdatei, welcher die Möglichkeiten zur Konfiguration der Menüleiste, der textuellen Ansicht und der Graphansicht enthält. Die Konfigurationsdatei muss als `properties.json` in dem jeweiligen `PROJEKT_ORDNER` abgelegt werden.

JSON

```
{
  "menu": {
    "defaultFile" : "textFiles/Merged.od",
    "showAllDropdownElements" : false,
    "dropdownElements" : ["JavaArtifactData.od", "Merged.od"]
  },
  "textual": {
    "extensions" : ["od", "txt"]
  },
  "graph" : {
    "tree" : {
      "Artifacts" : {
        "elements" : ["Directory", "JavaSourceFile",
          "FreemarkerTemplate"],
        "containsLinks" : ["contains"]
      }
    },
    "hierarchy" : {
      "elements" : ["Directory", "JavaSourceFile",
        "FreemarkerTemplate"],
      "containsLinks" : ["contains"],
      "links" : ["reliesOnTemplate", "reliesOnJavaArtifact",
        "imports"]
    }
  }
}
```

Listing 9.10: Projektspezifische Einstellungen zur Konfiguration der Menüleiste, der textuellen Ansicht und der Graphansicht mittels `properties.json`

Zunächst lässt sich die Artefaktdatendatei konfigurieren, welche initial angezeigt werden soll (`defaultFile`). Außerdem lässt sich einstellen, ob sich alle verfügbaren OD-Dateien im Dropdown Menü von ArtViz auswählen lassen sollen. Voraussetzung ist hier, dass sie sowohl im OD-Format als auch im JSON-Format vorliegen. Sollen alle Elemente angezeigt werden (`"showAllDropdownElements" : true`), kann der Eintrag `dropdownEle-`

ments weggelassen werden. Die nächsten beiden Einträge konfigurieren die textuelle Ansicht (textual) und die Graphansicht (graph). Durch die Angabe bestimmter Dateiendungen (extensions) lässt sich die Auswahl der angezeigten Dateien einschränken. Somit ist es möglich, die intern genutzten JSON-Dateien auszublenden. Im Eintrag tree lassen sich Tabs zur Baumdarstellung hinzufügen. Durch elements lassen sich die anzuzeigenden Elemente der Baumdarstellung konfigurieren. Zusätzlich müssen die hierarchischen Beziehungen (containsLinks) definiert werden. Die gleichen Einstellungen lassen sich auch für die hierarchische Graphdarstellung (hierarchy) angeben. Hier müssen jedoch auch die Arten der Links spezifiziert werden, die zu Querverbindungen im Graphen führen (links).

Ein weiterer Unterordner des PROJEKT_ORDNERS wird benötigt, um die Ergebnisse der Metrikberechnungen anzuzeigen. In diesen Ordner mit dem Namen metrics sind die Metrikdaten als JSON-Datei abzulegen. Die Darstellung der Metriken in ArtViz lässt sich ebenfalls über die properties.json Datei konfigurieren. Listing 9.11 zeigt einen vereinfachten Ausschnitt der Konfigurationsdatei.

```
{                                                                    JSON
  ...
  "metrics" : {
    "Efferente Kopplung": {
      "graphic" : ["Horizontal Bar Chart", "Vertical Bar Chart"],
      "defaultGraphic": "Horizontal Bar Chart",
      "description": "Anzahl reliesOn Beziehung zu anderen
        Artefakten"
    },
    "Afferente Kopplung" : {
      "graphic" : ["Horizontal Bar Chart", "Vertical Bar Chart"],
      "defaultGraphic" : "Horizontal Bar Chart",
      "description" : "Anzahl anderer Artefakte mit reliesOn
         Beziehung zu diesem Artefakt"
    },
    "LoC (Template)" : {
      "graphic" : ["Pie Chart"],
      "defaultGraphic" : "Pie Chart",
      "description" : "Anzahl Codezeilen"
    }
  }
}
```

Listing 9.11: Konfiguration von Metriken per properties.json Datei.

Für jede Metrik wird die graphische Darstellung separat konfiguriert. graphic spezifiziert dabei, welche Darstellungsarten auswählbar sind, während defaultGraphic die initiale

Darstellungsform definiert. Zusätzlich kann eine Beschreibung angegeben werden, die später per Hilfetext angezeigt werden kann.

Durch Angabe dieser Konfigurationen und die entsprechende Ablage der Artefaktdaten im `PROJEKT_ORDNER` lässt sich ArtViz vollständig nutzen. Eine Übersicht über die benötigte Ordnerstruktur ist in Listing 9.12 dargestellt.

```
assets
+-- globalProperties.json
+-- PROJECT_FOLDER_1
|   +-- metricFiles
|   |   +-- metrics.json
|   +-- properties.json
|   +-- textFiles
|       +-- JavaArtifactData.json
|       +-- JavaArtifactData.od
|       +-- Merged.json
|       +-- Merged.od
+-- PROJECT_FOLDER_2
    +-- properties.json
    +-- textFiles
        +-- DescriptiveArchitecture.json
        +-- DescriptiveArchitecture.od
        +-- DifferenceArchitecture.json
        +-- DifferenceArchitecture.od
        +-- PrescriptiveArchitecture.json
        +-- PrescriptiveArchitecture.od
```

Listing 9.12: Die Ordnerstruktur des `assets`-Ordners von ArtViz.

ArtViz wird derzeit aktiv weiterentwickelt. Es ist geplant, die verpflichtende Angabe von Einstellungen weiter zu reduzieren, um einen möglichst einfachen Einstieg für neue Benutzer zu schaffen. So sollen etwa die hierarchische Graphdarstellung und die Baumdarstellung auch ohne Angabe weiterer Konfigurationen Artefaktdaten anzeigen können. Die verwendeten hierarchischen ODs (vgl. Abschnitt 2.4.19s beinhalten bereits eine Aufteilung in die verschiedenen Linkarten, welche zu diesem Zweck genutzt werden kann.

Um die Verwendung noch weiter zu vereinfachen, müsste das Anlegen neuer Projekte und das Hochladen der Daten über die graphische Oberfläche zulässig sein. Auch sollten Benutzer dann nur noch die für sie relevanten Projekte einsehen können. Dazu müssten sie sich bei Aufruf der Webseite authentifizieren. Um eine solche Benutzerverwaltung, einen sicheren Datenzugriff, das Hochladen von Daten und die projektspezifischen Konfigurationen der Webseite zur Laufzeit zu ermöglichen, sollte eine Datenbank angebunden werden. Weiterhin muss die entsprechende Funktionalität auch in der graphischen Oberfläche der Webseite implementiert werden. Dort gibt es aktuell noch keine Möglichkeit zum Login, Datenupload, oder zur Konfiguration

der einzelnen Ansichten. Eine Benutzerverwaltung könnte dabei analog zu dem in [Her14] vorgestellten Ansatz implementiert werden.

9.5. Einbinden von MontiArT in eine Software-Engineering-Infrastruktur

In diesem Abschnitt wird gezeigt, wie sich Analysen mit MontiArT automatisiert starten, ausführen und wiederholen lassen, sodass die aktuellen Ergebnisse direkt in ArtViz einsehbar sind. Vorausgesetzt wird eine gewisse IT-Infrastruktur für das Software Engineering. Die Ergebnisse einer Analyse lassen sich damit automatisch an ArtViz übermitteln, wodurch sie in aktueller und adäquater Form zur Verfügung stehen. Somit lassen sich durch die hier vorgestellte Einbindung der Werkzeugkette in eine Software-Engineering-Infrastruktur die Anforderungen *TRQ-4.2* und *TRQ-4.3* erfüllen, welche fordern, dass vorgenommenen Analysen automatisiert, basierend auf aktuelle Artefaktdaten durchgeführt werden und ihre Ergebnisse in adäquater Form und einfach zugänglich zur Verfügung stehen.

9.5.1. Software-Engineering-Infrastruktur

Für das Einbinden der Werkzeugkette wird eine Software-Engineering-Infrastruktur benötigt, die in diesem Fall aus drei miteinander verknüpften Infrastrukturkomponenten besteht, deren Funktionsweise und Interaktion im Folgenden erläutert werden.

Versionskontrollsysteme (engl.: *Version Control System* (VCS)) speichern Dateien sowie die Änderungen an den Dateien über die Zeit hinweg. Der Zugriff auf verschiedene Dateiversionen ist somit jederzeit möglich. Dies bietet den Vorteil, dass sich ungewollte Änderungen rückgängig machen und sich verschiedene Versionen der Dateien miteinander vergleichen lassen. Zusätzlich bieten VCSs die Möglichkeit verschiedene Entwicklungszweige zu erstellen, die ausgehend von einer bestimmten Version unabhängig voneinander weiterentwickelt werden können. Darüber hinaus wird üblicherweise Unterstützung bei der Zusammenführung von Entwicklungszweigen angeboten. Bekannte Beispiele für VCSs aus dem Bereich der Softwareentwicklung sind SVN [PCSF11] und Git [CS14]. Die Idee findet aber mittlerweile auch außerhalb der reinen Informatik Einzug. So wird beispielsweise bei Verwendung von Office 365 [Mur11]) bei Ablage der Daten in OneDrive die Versionshistorie ebenfalls mitgespeichert.

Artifact Repository Manager oder auch *Binary Repository Manager* speichern wiederverwendbare Softwarekomponenten z.B. in Form von Bibliotheken oder Frameworks. Eigene Komponenten lassen sich so einfach über das Entwicklerteam verteilen. Externe Komponenten lassen sich zwischenspeichern und somit schneller verfügbar machen. Außerdem kann der Zugriff auf externe Komponenten kontrolliert werden. Im Gegensatz zu VCSs werden Artifact Repository Manager für bereits gepackte Komponenten genutzt. Dies umfasst sowohl Quellcode als auch ausführbare Binaries. Zwei prominente Beispiele für Artifact Repositories sind Nexus [MW16] und Artifactory [JFr17].

Die dritte Art von betrachteten Infrastrukturkomponenten sind Server, die eine kontinuierliche Integration (engl. *Continous Integration* (CI)) von entwickelten Softwareanteilen unterstützen. Fowler definiert CI wie folgt: *„Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.“* [Fow06]

Bei CI wird häufig ein zentraler Buildprozess angestoßen, um sicherzustellen, dass sich das integrierte Produkt wie gewünscht erstellen lässt. Hierzu wird ein zentraler CI-Server benötigt, welcher den Buildprozess nach einer Integration durchführt. Beispiele für solche Server sind Jenkins [Sma11], Travis [Tra17], TeamCity [Mel12] und Bamboo [Atl17].

Die drei Infrasturkturkomponenten lassen sich verknüpfen, wodurch ihr Nutzen wesentlich erhöht werden kann. Der CI-Server kann auf das VCS zugreifen, wodurch er automatisch die neueste Version des Software-Engineering-Projekts vorliegen hat. Ferner lässt sich durch eine regelmäßige Prüfung des VCS bei Änderungen im Projekt automatisch ein Build starten. Die für den Build benötigten und im Artifact Repository Manager gespeicherten Artefakte lassen sich dann, genau wie bei Durchführung des Builds auf einer lokalen Entwicklermaschine, automatisch abrufen.

9.5.2. Einbinden der Werkzeugkette

Mit Hilfe der vorgestellten Infrastrukturkomponenten lässt sich MontiArT so einbinden, dass bei jeder Änderung im Projekt automatisch die gewünschten Analysen durchgeführt und die Ergebnisse der Analysen in ArtViz zur Verfügung gestellt werden. Abbildung 9.13 zeigt eine Übersicht über den Ablauf des Prozesses unter Verwendung der Software-Engineering-Infrastruktur mit den beteiligten Rollen am Beispiel des DEx-Projekts. Bei der vorgestellten Einbindung der Werkzeugkette wurden SVN als VCS, Jenkins als CI-Server und Nexus als Artifact Repository Manager verwendet.

Die Integration von Änderungen an dem Projekt durch einen *Entwickler* bzw. *Werkzeugentwickler* werden in Schritt (1) durch ein Commit in das VCS durchgeführt. Als nächstes wird auf dem CI-Server ein entsprechender Buildprozess ausgeführt. Buildprozesse werden bei Jenkins in *Jobs* gekapselt. Die Bedingung zum Starten eines Buildprozess lässt sich pro Job frei konfigurieren. In diesem Fall werden zwei verschiedene Jobs benötigt. Der Erste Job führt den Standard Buildprozess des Projekts durch. Er wird ausgelöst sobald eine Änderung im VCS erkannt wird. Ist dies der Fall, wird in Schritt (2) der aktuelle Stand aus dem VCS abgeholt und der Job ausgeführt. Nach der Durchführung stehen die Quellen, die kompilierten Artefakte des Produkts und die gepackten Produkte in dem *Workspace* des CI-Jobs zur Verfügung. Der zweite Job des CI-Servers ist für die MontiArT-Analyse verantwortlich. Dieser Job wird automatisch nach Beendigung des ersten Jobs angestoßen. Als erste Aktion seiner Ausführung wird, wie durch Schritt (3) dargestellt, der Workspace kopiert. Bevor die Analysen ausgeführt werden können, werden die Werkzeugkettenkonfiguration sowie manuell erstellte Eingaben für die Analyse (Schritt (4)) aus dem VCS abgerufen. Weiterhin werden die zur Durchführung

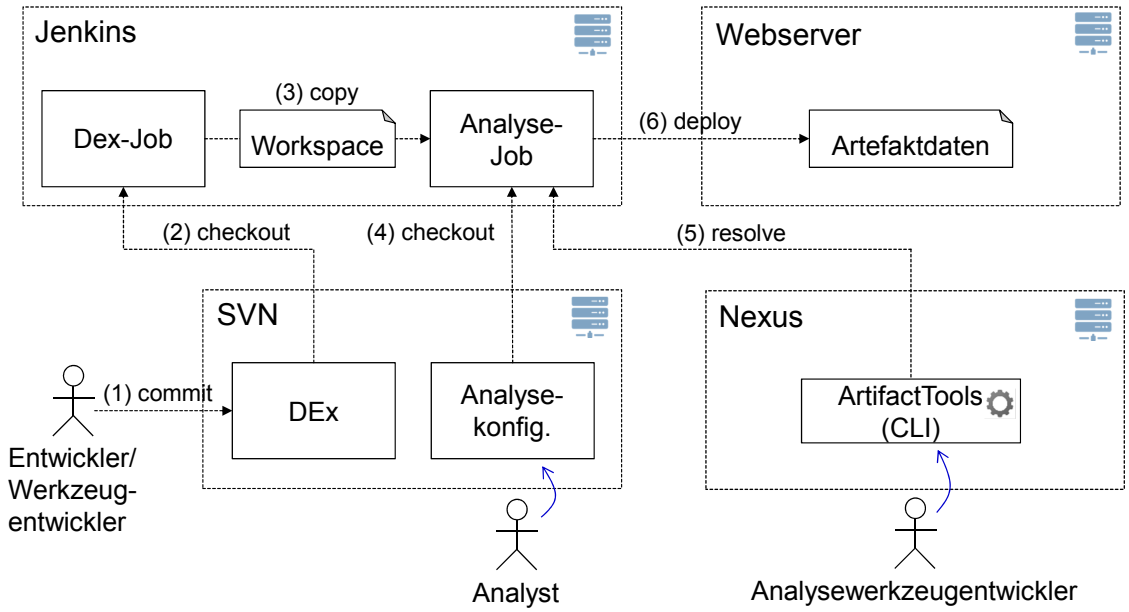

Abbildung 9.13.: Integration der Artefaktdatenanalyse in die bestehende Infrastruktur.

der Analysen benötigten Analysewerkzeuge von dem Artifact Repository Manager abgerufen (Schritt (5)). Anschließend wird die Analyse durchgeführt und abschließend die Ergebnisse an den ArtViz-Webserver ausgeliefert (Schritt (6)), wodurch die aktuellen Analysedaten direkt im Anschluss zur Verfügung stehen. Die gezeigte Einbindung erfüllt somit die Anforderungen *TRQ-4.2* und *TRQ-4.3*, da die Analysen bei einer Integration von Änderungen automatisch durchgeführt und deren Ergebnisse in ArtViz für alle beteiligten Stakeholder zur Verfügung gestellt werden.

Bei der vorgestellten Einbindung kann der Analysejob so konfiguriert werden, dass er bei Änderungen an den Analyseskripten durch den *Analysten* ebenfalls automatisch gestartet wird. Änderungen an den Analysewerkzeugen durch den *Analysewerkzeugentwickler* können hingegen nicht automatisch erkannt werden. In solchen Fällen muss der Analysejob deshalb von Hand gestartet werden.

Kapitel 10.

Artefaktbasierte Analysen in der Anwendung

In diesem Kapitel werden artefaktbasierte Analysen von vier Projekten des Lehrstuhls für Software Engineering der RWTH Aachen University vorgestellt. Dabei werden in allen Fällen die relevanten Artefakt- und Beziehungsarten unter Verwendung des in Kapitel 6 vorgestellten Artefaktmodells beschrieben und, wo nötig, zusätzliche Erweiterungen oder Änderungen zu Analysezwecken behandelt. Die Analysen selbst werden per OCL spezifiziert oder es werden die bereits vorgestellten Analysen aus Abschnitt 6.6 angewendet. Darüber hinaus werden projektspezifische Extraktionswerkzeuge sowie die zugehörigen Analyseergebnisse vorgestellt.

10.1. Der DEx-Generator

Der in Abschnitt 2.4.3 vorgestellte Generator wird zusammen mit einem Modell zur Generierung einer Applikation für ein soziales Netzwerk ausgeliefert. Mit diesem Modell als Eingabe lässt sich der DEx-Generator direkt ausführen und somit ein auf dem Modell basierendes Produkt generieren. Darüber hinaus wurde zusammen mit dem Architekten des Projekts die in Abbildung 10.1 dargestellte Soll-Architektur erstellt. Die Architektur besteht aus elf Modulen. Diese wurden zunächst ihrem Verwendungszweck innerhalb des Projekts zugeordnet. Module können Teil des Generators, des generierten Produkts oder der Laufzeitumgebung sein. Zusätzlich wurden die Module einer technischen Schicht zugeordnet. Dabei sind in Ausnahmefällen auch Zugriffe auf oder Beiträge zu einer darüber liegenden Schicht erlaubt. Die Module `gtr-commons` und `rte-util` stellen Hilfsklassen bereit, die von Modulen aller Schichten genutzt werden dürfen.

Im Folgenden wird als Analyse die Ist-Architektur und die Differenzarchitektur zwischen erhobener Ist- und dargestellter Soll-Architektur vorgestellt. Als Analysegegenstand wurde die Version 4.0.0 des Projekts verwendet. Die vorgestellte Analyse wurde ebenfalls in den Arbeiten [Hil16, The17] basierend auf der hier vorgestellten Soll-Architektur aber mit einer älteren Version des AMs und ohne Unterscheidung verschiedener Modulbeziehungsarten durchgeführt.

Bei der Durchführung der Analyse müssen im ersten Teilschritt zunächst die relevanten Beziehungen zwischen den Artefakten extrahiert werden. Neben den mitgelieferten Extrakto-

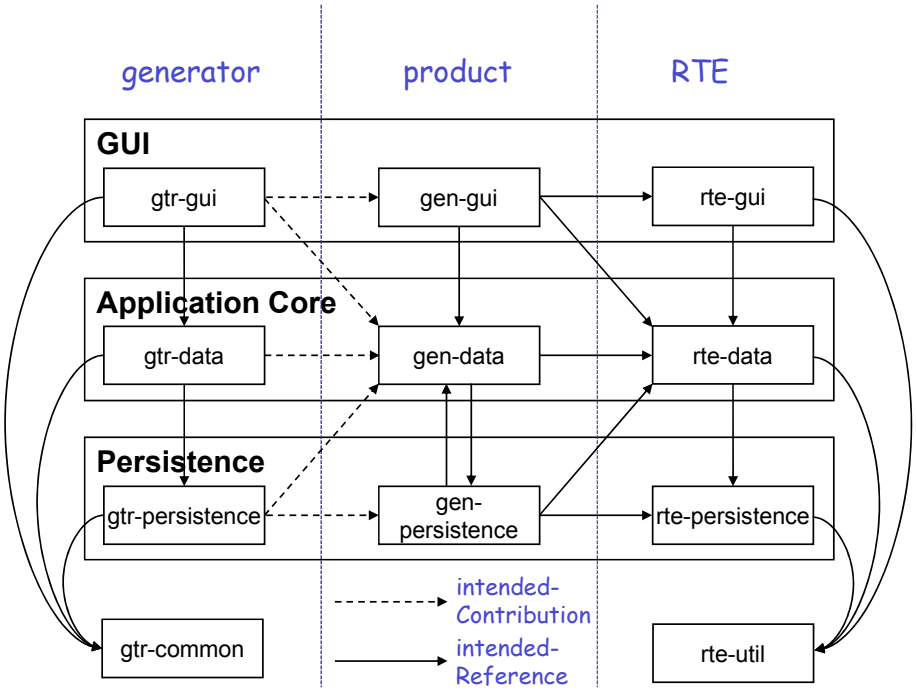

Abbildung 10.1.: Die Soll-Architektur des DEx-Projekts.

ren wurden auch zwei projektspezifische Extraktoren implementiert und eingebunden. Einer der Extraktoren ist ein dynamischer Extraktor, der nach der Methodik aus Abschnitt 9.2.3 implementiert wurde.

Die beiden projektspezifischen Extraktoren werden in Abschnitten 10.1.1 und 10.1.2 vorgestellt, bevor die Durchführung der Analyse und die Analyseergebnisse in Abschnitt 10.1.3 diskutiert werden.

10.1.1. DEx Transformation Extractor

Neben den bereits in Abschnitt 2.4.3 beschriebenen Funktionen bietet der DEx-Generator weitere Konfigurationsmöglichkeiten. So ist es beispielsweise möglich, die von dem Generator genutzten Templates nachträglich auszutauschen. Dazu werden in Java-Transformationen geschrieben, welche zum einen M2M-Transformationen auf dem AST durchführen und zum anderen für bestimmten AST-Knoten das vorgesehene Standardtemplate durch ein spezifisches Template ersetzen (siehe *Template Attachment* [Rot17]).

Beispielsweise ist `EmptyMethod.ftl` das Standardtemplate für Knoten des Typs `AST-CDMethod`, welches einen leeren Methodenrumpf generiert. Um für einen einzelnen Knoten dieses Typs einen spezifischen Methodenrumpf zu generieren, kann das Standardtemplate ausgetauscht werden. Der in Listing 10.2 gezeigte, vereinfachte Methodenaufruf ersetzt das Standardtemplate für den in der Variable `ast` gespeicherten AST-Knoten durch das Template mit dem Namen `SpecificTemplate`.

```
1  replaceTemplate(CoreTemplate.EMPTY_METHOD, ast,                    Java
2    new TemplateHookPoint("custom.SpecificTemplate"));
```

Listing 10.2: Exemplarischer Aufruf der Methode `replaceTemplate` zum Ersetzen von Templates in DEx.

Die Namen der Templates werden in DEx entweder direkt oder durch Nutzung von Konstanten, die in den beiden Java-Enums `CoreTemplate` und `AssociationMethodFactory.Templates` gekapselt sind, angegeben. Im oben gezeigten Beispiel wurde mit `custom.SpecificTemplate` ein Templatename direkt angegeben und mit `CoreTemplate.EMPTY_METHOD` eine Konstante aus einem Enum verwendet.

Das Ziel des `DExTransformationExtractors` ist es, Beziehungen dieser Art zwischen den Java-Quelldateien, welche die Methodenaufrufe enthalten, und den Templatedateien zu extrahieren. Dabei wird pro Methodenaufruf jeweils ein Link für das ersetzte und ein Link für das ersetzende Template erstellt. Somit extrahiert der `DExTransformationExtractor` zu den in Tabelle 10.3 dargestellten Anteilen des AMs zugehörige Artefaktdaten.

Klasse	**Attribut**	**Assoziation**
JavaSourceFile	simpleName fullName	reliesOnTemplate
FreeMarkerTemplate	simpleName fullName	

Tabelle 10.3.: Durch den `JavaExtractor` extrahierte AM-Anteile.

Zur Extraktion dieser Verknüpfungen führt der Extraktor die folgenden Schritte aus:

1. Laden aller potentiell in Frage kommenden Template- und Java-Dateien, d.h., Template- und Java-Dateien liegen in dem als Parameter übergebenen *Eingabefpad* (siehe Abschnitt 8.2). Relevante Java-Dateien erben von der abstrakten Klasse `DexTransformation`.

2. Durchsuchen aller potentiellen Java-Dateien. Für jedes gefundenen `replaceTemplate`- bzw. `new TemplateHookPoint`-Statement:

   a) Nachschlagen des zugehörigen Templatenames, falls eine Enumkonstante zur Referenzierung eines Templates genutzt wurde.

   b) Suchen der zugehörigen Templates anhand ihrer Namen in der Menge der geladenen Templates.

3. Ausgabe aller involvierten Artefakte und Beziehungen im Artefaktdatenformat. Dies umfasst alle Java-Klassen, von denen eine solche Verknüpfung ausgeht, alle verknüpften Templates und die Verknüpfungen selbst.

Das Ergebnis der Extraktion für das Beispiel aus Listing 10.2 ist in Listing 10.4 abgebildet. Es impliziert, dass die entsprechenden Artefakte existieren und von dem Extraktor gefunden werden konnten. Die Ausgabe folgt den Konventionen der Extraktoren die in Abschnitt 8.2.1 beschrieben wurden.

OD
«ArtData»

```
objectdiagram ReplaceTemplateArtifactData {

  @MethodTransformer!java:JavaSourceFile {
    simpleName = "MethodTransformer";
    nameExtension = "java";
    fullName = "custom/MethodTransformer.java";
  };

  @EmptyMethod!ftl:FreeMarkerTemplate{
    simpleName = "EmptyMethod";
    nameExtension = "ftl";
    fullName = "core/EmptyMethod.ftl";
  };

  @SpecificTemplate!ftl:FreeMarkerTemplate {
    simpleName = "SpecificTemplate";
    nameExtension = "ftl";
    fullName = "custom/SpecificTemplate.ftl";
  };

  link reliesOnTemplate
    @MethodTransformer!java -> @SpecificTemplate!ftl ;

  link reliesOnTemplate
    @MethodTransformer!java -> @EmptyMethod!ftl ;
}
```

Listing 10.4: Ergebnis der Extraktion für den exemplarischer Methodenaufruf aus Listing 10.2.

10.1.2. DEx Action Reporter

Der `DExActionReporter` ist ein dynamischer Extraktor, der speziell zur Analyse von DEx-Projekten entwickelt wurde. Dieses Werkzeug ermöglicht es, `Actions`, die während des Ablaufs des DEx-Generators auftreten, im Artefaktdatenformat auszugeben. Der Extraktor wurde basierend auf der in Abschnitt 9.2.3 vorgestellten Methodik erstellt. Da der DEx-Generator im Gegensatz zu dem in Abschnitt 6.5 beschriebenen Ablauf von MontiCore-basierten Generatoren einige Besonderheiten aufweist, musste ein speziell auf DEx-Projekte zugeschnittener Extraktor entwickelt werden. Neben der Extraktion der Aktionen, genutzten

und erstellten Artefakten sowie deren Beziehungen werden auch die abgeleiteten Eigenschaften `contributionFT` und `contributionJava` direkt von dem Extraktor mitberechnet.

Als dynamischer Extraktor wird der `DExActionReporter` nicht in einem eigenen Jar-Archiv ausgeliefert, sondern wurde als eine Klasse des DEx-Generators nachträglich hinzugefügt und, wie in Abschnitt 9.2.3 beschrieben, als `IReportEventHandler` registriert. Listing 10.5 zeigt die Änderungen des in diesem Anwendungsfall verwendeten AMs gegenüber dem in Kapitel 6 vorgestellten Modell. Die Gründe für die Anpassungen werden im Folgenden erläutert.

CD «AM»

```
1  class GenerationAction extends Action {}
2
3  context GenerationAction inv:
4    contributesTo == sub.contributesTo;
5
6  context TemplateExecution inv:
7    contributesTo == parent**.contributesTo;
8
9  context JavaMethodCall inv:
10   contributesTo == parent**.contributesTo;
11
12 association /contributionJava
13   [*] JavaSourceFile -> Artifact [*];
14
15 context JavaSourceFile inv:
16   contributionJava == compiledTo.contributionJava;
```

Listing 10.5: Änderung der Berechnungsvorschrift für die `contibutesTo` Beziehung für `JavaMethodCalls`.

Die erste Besonderheit besteht darin, dass der DEx-Generator stark auf vorbereitenden M2M-Transformationen aufbaut. Ein Großteil der Logik des Werkzeugs wurde mit Hilfe der Transformationen umgesetzt, sodass der abschließende Generierungsschritt eher leichtgewichtig ist. Der resultierende AST, der als Eingabe für den Generierungsschritt dient, enthält auf oberster Ebene AST-Knoten der Typen `ASTCDClass`, `ASTCDInterface` und `ASTCDEnum`. Für jeden dieser Knoten erfolgt ein eigener Generierungsaufruf mit einem passenden Starttemplate, der zur Generierung von jeweils genau einem Artefakt führt. Aus diesem Grund protokolliert der `DExActionReporter` eine eigene `GenerationAction` für jeden dieser Aufrufe.

Eine weitere Besonderheit ist, dass die Templates zuerst vollständig durchlaufen werden und erst nach der Berechnung des konkreten Artefaktinhalts ein entsprechendes Artefakt erstellt und der Inhalt in das Artefakt geschrieben wird. Aus diesem Grund ist die `CreateAction` keine `subaction` einer `TemplateExecution`, sondern eine `subaction` der `GenerationAction`, die der Bearbeitung des aktuellen AST-Knotens entspricht. Somit lässt sich die

contributesTo-Beziehung von GenerationActions anhand deren subactions berechnen.

Zusätzlich ist es unter Verwendung der aktuellen Reportinginfrastruktur nicht möglich, zu entscheiden, ob während der zugehörigen TemplateExecution in das Zielartefakt geschrieben oder ob das Template lediglich als Strukturierungshilfe eingesetzt wurde. Aus diesem Grund werden keine Daten für die writesTo-Beziehung (vgl. Abschnitt 6.5.2) extrahiert, sondern zur Berechnung der contributesTo-Beziehungen von Templates auf die contributesTo-Beziehung der entsprechenden GenerationAction zurückgegriffen.

Eine ähnliche Anpassung musste ebenfalls für JavaMethodCalls umgesetzt werden. Die aktuelle Reportinginfrastruktur meldet nur bei der Instanziierung einer Java-Klasse ein Event. Andere Methodenaufrufe werden hingegen nicht gemeldet. Darüber hinaus gibt es auch keine Möglichkeit mittels Reportinginfrastruktur herauszufinden, ob ein per Java-Klasse berechneter Wert anschließend als Ausgabe in ein Artefakt geschrieben wird, weshalb auch die Berechnung der contributesTo-Beziehung für JavaMethodCalls entsprechend angepasst wurde. Dies führt dazu, dass eine Java-Klasse auch zur Erstellung des zugehörigen Zielartefakts beiträgt, wenn diese während einer GenerationAction instanziiert wurde. Diese Beziehungen werden durch die contributionJava-Assoziation des AMs aus Abschnitt 6.1 repräsentiert. Da in der späteren Analyse JavaSourceFiles anstelle von JavaClassFiles untersucht werden sollen, wurde eine solche Assoziation auch für JavaSourceFiles definiert. Beziehungen dieser Art werden einfachheitshalber ebenfalls direkt vom DExActionReporter mit erhoben.

Tabelle 10.6 fasst die durch den DExActionReporter extrahierten und zu Anteilen des angepassten AMs zugehörigen Artefaktdaten zusammen.

Klasse	**Attribut**	**Assoziation**
FreeMarkerTemplate	simpleName fullName	contributionFT
JavaSourceFile	simpleName fullName	contributionJava
Action		subaction
GernerationAction		
ArtifactCreate		creates
JavaMethodCall		file
TemplateExecution		template

Tabelle 10.6.: Durch den DExActionReporter extrahierte AM-Anteile.

Dabei ist zu beachten, dass nur Artefaktdaten zu den folgenden Subklassen der Action-Klasse extrahiert werden: ArtifactCreate, GenerationAction, JavaMethodCall, und TemplateExecution .

10.1.3. Durchführung der Analyse

In diesem Abschnitt wird die Durchführung der Analyse beschrieben. Die vollständige Definition der erstellten MontiArT-Instanz, welche die Verknüpfung der verschiedenen Werkzeuge zur Analyse des DEx-Projekts umfasst, ist in Anhang F.1 in Form eines ausführbaren Skripts dargestellt.

Zur Extraktion der Daten werden die drei in Abschnitt 8.2 beschriebenen Standardextraktoren `ArtifactContainerExtractor`, `JavaExtractor` und `TemplateExtractor` verwendet. Zusätzlich werden die beiden in den vorherigen Abschnitten vorgestellten Extraktoren `DExTransformationExtractor` und `DExActionReporter` in die MontiArT-Instanz eingebunden. Da der `DExActionReporter` direkt die für die Analyse benötigten abgeleiteten Eigenschaften bereitstellt, müssen diese nicht mehr in einem späteren Schritt berechnet werden. Stattdessen werden mit Hilfe des `ODFilters` (siehe Abschnitt 8.5.3) alle nicht für die Analyse benötigten Artefaktdaten entfernt, sodass nur Artefakte und Beziehungen zwischen Artefakten übrig bleiben. Im Fall des `DExActionReporters` werden `Actions` und die zugehörigen Beziehungen, im Falle des `JavaExtractors` `Packages` und `Typen` sowie deren Beziehungen entfernt.

Im darauffolgenden Schritt werden die Artefaktdaten mittels `ODMerger` zu einer einzigen Artefaktdatendatei zusammengefügt, welche nun eine der Eingaben für die Berechnung der Ist-Architektur darstellt.

Als zweite Eingabe für die Berechnung der Ist-Architektur wird die *Moduldefinition*, also die Zuordnung von Artefakten zu den Modulen der Soll-Architektur aus Abbildung 10.1, benötigt. Eine entsprechende Datei wurde, wie in Abschnitt 7.2.3 beschrieben, erstellt. Tabelle 10.7 zeigt die in der Moduldefinition angegebene Zuordnung von Artefakten zu Modulen.

Module	**Ordner**
gtr-gui	gtr/src/cd2swing
gtr-data	gtr/src/cd2data
gtr-persistence	gtr/src/cd2persistence
gtr-commons	gtr/src/common
gen-gui	use/gen/dex/socnet/gui
gen-data	use/gen/dex/socnet (ohne Unterordner)
gen-persistence	use/gen/dex/socnet/persistence
rte-gui	rte/src/dex/gui
	rte/src/dex/search
	rte/src/dex/help
rte-data	rte/src/dex/data
rte-persistence	rte/src/dex/persistence
	rte/src/dex/server/access
rte-util	rte/src/util

Tabelle 10.7.: Moduldefinition des DEx-Projekts: Zuordnung von Ordnern zu Modulen.

Die vollständige Definition im OD-Format lässt sich Anhang F.2 entnehmen. Dabei wurden alle Dateien des `gen-data`-Modules einzeln zugewiesen, da sie zusammen mit Ordnern anderer Module (`use/gen/dex/socnet/gui` und `use/gen/dex/socnet/persistence`) in einen gemeinsamen Ordner `use/gen/dex/socnet` generiert werden. Zusätzlich wurde in der in Anhang F.2 gezeigten Datei auch die Soll-Architektur spezifiziert. Die Spezifikation in einer einzigen Datei hat den Vorteil, dass die Module nicht doppelt in verschiedenen Modelldateien angegeben werden müssen. Mit Hilfe des `ODFilters` lassen sich die Moduldefinition und die Soll-Architektur ableiten und in verschiedenen Artefakten speichern.

Die so erstellte Moduldefinition wird anschließend mit den Artefaktdaten zusammengeführt und dem `ArcReconstructor` als Eingabe übergeben. Dieser berechnet basierend auf dem übergebenen Datensatz die Ist-Architektur. In einem Zwischenschritt werden dabei, wie in Abschnitt 8.5.4 beschrieben, alle Artefakte, die sich innerhalb der zugeordneten Ordner befinden (dabei werden auch alle Unterordner rekursiv betrachtet), dem Modul zugeordnet. Das Ergebnis der Berechnung ist graphisch in Abbildung 10.8 dargestellt. Das Ergebnis im ursprünglichen OD-Format ist in Anhang F.3 zu finden.

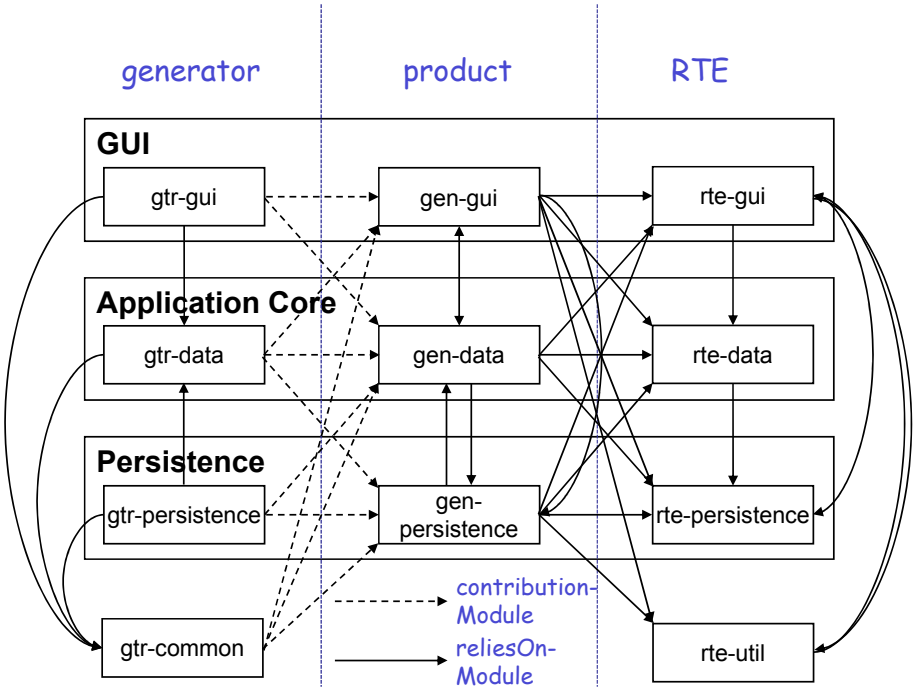

Abbildung 10.8.: Die Ist-Architektur des DEx-Projekts.

Basierend auf der berechneten Ist-Architektur und der modellierten Soll-Architektur lässt sich nun die Differenzarchitektur bestimmen. Dazu wurde der `ACChecker` von MontiArT verwendet, welcher sowohl die Soll-Architektur als auch die Ist-Architektur als Eingabe bekommt und die Differenzarchitektur berechnet. Die Differenzarchitektur des DEx-Projekts ist in Abbildung 10.9 dargestellt und enthält sowohl `missingReference`-Beziehungen als

auch `unintendedReference`- und `unintendedContribution`-Beziehungen. Die Differenzarchitektur im ursprünglichen OD-Format ist in Anhang F.4 dargestellt.

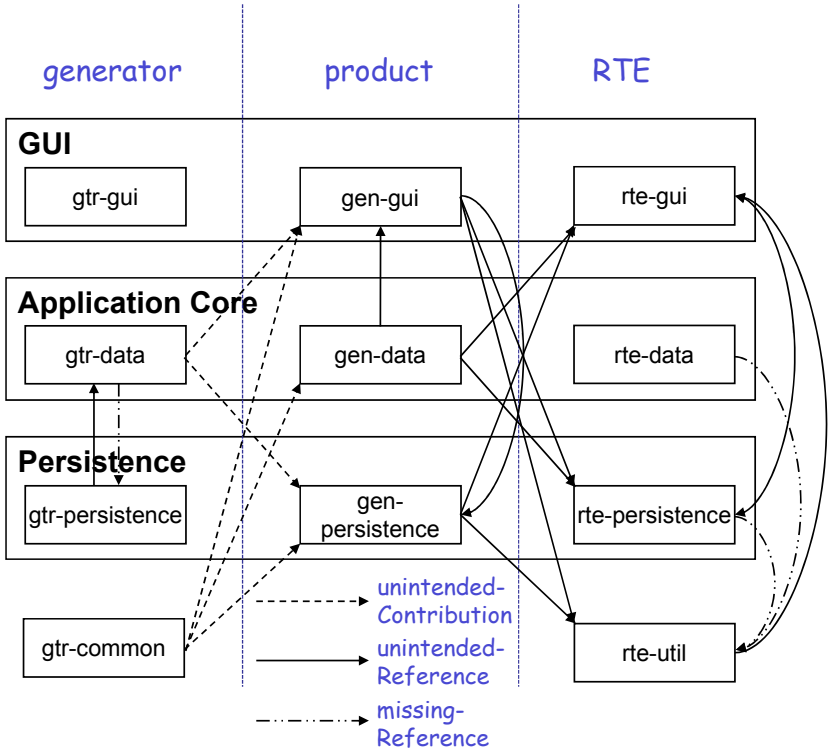

Abbildung 10.9.: Die Differenzarchitektur des DEx-Projekts.

Die Anzahl der Beziehungen der Differenzarchitektur macht deutlich, dass bereits eine größerer Architekturdrift stattgefunden hat. Architekturdrift führt nach [Lil16] dazu, dass pro Zeiteinheit immer weniger Funktionalität implementiert und Bugs behoben werden können, was letztlich zu frustrierten und demotivierten Entwicklern führt. Basierend auf der durchgeführte Analyse lässt sich der Architekturdrift eliminieren, wodurch diesen Problemen entgegen gewirkt werden kann. Dazu muss für jede Beziehung einzeln entschieden werden, ob es sich dabei, um eine Verletzung der gewollten Soll-Architektur handelt oder ob die Soll-Architektur an dieser Stelle veraltet ist. Die Ursache für ungewollte Beziehungen der Differenzarchitektur lassen sich durch die vom `ArcReconstructor` als Nebenprodukt erstellte Reporting-Datei nachvollziehen, wodurch diese Arbeit erleichtert wird.

Ist die Soll-Architektur für eine der Beziehungen nicht mehr aktuell, so muss sie angepasst werden. Dazu müssen fehlende geforderte Beziehungen in der Soll-Architektur hinzugefügt und geforderte Beziehungen, die mittlerweile überflüssig geworden sind, entfernt werden. Ist hingegen die Soll-Architektur weiterhin gültig, so liegt ein Problem im Quellcode des Projekts vor. Bei dessen Behebung müssen fehlende und zusätzliche, ungewollte Beziehungen der Ist-Architektur unterschiedlich behandelt werden. Ungewollte Beziehung müssen durch ein Refactoring des Quellcodes eliminiert werden. Fehlende Beziehungen deuten hingegen auf

einen Fehler im Projekt hin, da sie in der aktuellen Soll-Architektur vorgesehen sind, in der Ist-Architektur aber fehlen. Eine Ausnahme hiervon ist der Fall, dass fehlende Beziehungen mit ungewollten Beziehung in Zusammenhang stehen. So könnte eine Schichtenarchitektur, bei der jede Schicht nur auf die darunterliegende Schicht zugreifen darf, verletzt werden, indem eine Schicht übersprungen wurde. In diesem Fall gibt es neben dem ungewollten Zugriff auch einen fehlenden Zugriff auf die darunterliegende Schicht. Eine solche Architekturverletzung muss nicht zwingend zu einem Fehlverhalten führen.

Abschließend werden die Zahlen zu der durchgeführte Analyse in den Tabellen 10.10, 10.11 und 10.12 zusammengefasst, um einen Eindruck von dem Umfang des untersuchten Projekts, der vorhandenen Artefakt- und Beziehungsarten sowie den beobachteten Aktionen zu vermitteln.

Module	**JavaSourceFile**	**FreeMarkerTemplate**	$\sum$
gtr-gui	19	96	115
gtr-data	27	92	119
gtr-persistence	14	93	107
gtr-commons	17	5	22
gen-gui	121	0	121
gen-data	44	0	44
gen-persistence	121	0	121
rte-gui	119	0	119
rte-data	10	0	10
rte-persistence	17	0	17
rte-util	8	0	8
Nicht zugewiesen	38	15	53
$\sum$	555	301	856

Tabelle 10.10.: Untersuchte Artefakte des DEx-Projekts

Beziehungsart	**Quellartefakt**	**Anzahl**
contains	Directory	940
imports	JavaSourceFile	1815
reliesOnJava	JavaSourceFile	3807
reliesOnTemplate	JavaSourceFile	324
reliesOnJava	FreeMarkerTemplate	88
reliesOnTemplate	FreeMarkerTemplate	59
contributionJava	JavaSourceFile	270
contributionFT	FreeMarkerTemplate	2760
$\sum$		10063

Tabelle 10.11.: Untersuchte Beziehungen zwischen Artefakten des DEx-Projekts.

Action	Anzahl
ArtifactCreate	287
GenerationAction	287
JavaMethodCall	681
TemplateExecution	7705
$\sum$	8960

Tabelle 10.12.: Beobachtete Aktionen während der Ausführung des DEx-Generators.

10.2. Eingabeartefakte des BumperBots

Der *BumperBot* [RRW13] ist eine mit der Sprachfamilie *MontiArcAutomaton* (MAA) modellierte Applikation. Die verwendeten Modelle beschreiben dabei sowohl die Softwarearchitektur als auch das Verhalten eines zugehörigen Roboters und dienen als Eingabe für einen Generator, der eine auf dem Roboter installierbare Applikation generiert. Der BumperBot wurde in zwei verschiedenen Varianten (A und B) modelliert. Die Modelle der Varianten wurden dabei in verschiedenen Packages abgelegt. Dies ermöglicht, dass beide Varianten innerhalb einer einzigen Analyse betrachtet werden können. Beide Varianten verwenden zur Modellierung der Applikation die gleiche Modellbibliothek. Variante A besteht aus zwei und Variante B aus 4 Modellartefakten. Die Bibliothek besteht aus 18 Artefakten, wovon 7 Modellartefakte und 11 Java-Dateien sind. Die Quellen des BumperBots sind als Teil von MA verfügbar[1]. In dieser Analyse wurde Version 4.1.5 von MA verwendet.

MAA-Artefakte enthalten sowohl die Softwarearchitektur (in Form von MA-Modellelementen) als auch die Verhaltensbeschreibung (in Form von Automaten). Für jede der Varianten existiert dabei ein MAA-Hauptartefakt, welches als Startpunkt für Analysen verwendet werden kann. CD-Artekfate werden zur Beschreibung von Datentypen verwendet. Darüber hinaus können sowohl simple Java-Datentypen als auch Java-Klassen der Pakete `java.lang` und `java.util` aus MAA-Artefakten heraus genutzt werden. Für relevante Klassen dieser Pakete wurden aus technischen Gründen Platzhalter in der Bibliothek hinzugefügt. Im Unterschied zu anderen Artefakten lassen sich diese Artefakte ohne zugehörige Imports verwenden.

Da sowohl der BumperBot in seinen beiden Varianten als auch die Modellbibliothek weiterentwickelt werden, ist die Kenntnis der Artefaktbeziehungen in diesem Projekt hilfreich. Mit Hilfe der artefaktbasierten Analyse lassen sich diese Beziehungen extrahieren. Weiterhin wird das Projekt auf ungenutzte Importe zwischen den Generatoreingabeartefakten und auf gänzlich ungenutzte Artefakte (ausgehend von den Hauptartefakten der beiden Varianten) hin untersucht, um vor der Weiterentwicklung eine möglichst saubere Modellbasis zu erhalten.

In einem ersten Schritt wurde das Artefaktmodell um die benötigten Elemente zur Analyse von MAA-Artefakte erweitert. Diese Erweiterungen sind in Listing 10.13 dargestellt.

Sowohl `imports`- als auch `typeUsage`-Beziehungen wurden ausgehend von `MAAModelFile`-Artefakten hinzugefügt. Eine `typeUsage`-Beziehung existiert genau dann, wenn ein

[1] https://git.rwth-aachen.de/montiarc/core

CD
«AM4A»

```
1  class MAAModelFile extends ModelFile {
2  }
3
4  association imports [*] MAAModelFile -> Artifact [*];
5  association typeUsage [*] MAAModelFile -> Artifact [*];
6  association /reliesOn [*] MAAModelFile -> Artifact [*];
7
8  context MAAModelFile inv:
9    reliesOn == typeUsage.addAll(imports);
10
11 context MAAModelFile inv:
12   refersTo.containsAll(reliesOn);
13
```

Listing 10.13: Die Erweiterung des AMs für die Eingabeartefakte des BumperBot-Projekts.

Typ aus einem anderen Artefakt zur Definition der Softwarearchitektur oder dem Verhalten genutzt wird. Ein Typ ist an dieser Stelle entweder ein Java-Typ, ein CD-Typ oder eine MA-Komponente. Durch die Vereinigung beider Beziehungen lassen sich die `reliesOn`- und die `refersTo`-Beziehungen berechnen.

Zur Extraktion zugehöriger Daten wurde ein eigener `MAAExtractor` implementiert. Dieser nutzt die Symboltabelle, um die Beziehungen zwischen Modell- und Java-Artefakten nach der in Abschnitt 9.2.2 beschriebene Methodik zu extrahieren. Dabei wird zur Bestimmung der beiden Beziehungsarten direkt auf der Datenstruktur der Symboltabelle gearbeitet. `imports`-Beziehungen lassen sich im `ArtifactScope` direkt ablesen während die `typeUsage`-Beziehung durch eine rekursive Suche in den `Scopes` der Symboltabelle (vgl. Abschnitt 2.3.2) erkannt werden. Beim Auflösen der Beziehungen müssen dann die möglichen Arten von Zieltypen durchprobiert werden, was zwar ineffizient, aber bei der geringen Anzahl verschiedener Zieltypen (MA-Komponente, CD-Typ oder Java-Typ) noch vertretbar ist.

Die zugehörigen extrahierten Artefaktdaten sind in Anhang G.1 dargestellt. Sie enthalten 24 Objekte, 24 `imports`-Links und 30 `typeUsage`-Links. Eine interessante Beobachtung ist, dass alle extrahierten Beziehungen von MAA-Artefakten ausgehen. Zur Realisierung der Datentypen mit Hilfe von CD-Artefakten lassen sich prinzipiell auch andere Artefakte wiederverwenden. Davon wurde bei der Modellierung an dieser Stelle jedoch kein Gebrauch gemacht. Die Java-Platzhalterartefakte haben ebenfalls keine ausgehenden Beziehungen.

Zunächst wurde in einer Analyse geprüft, ob alle importierten Artefakte auch wirklich in dem importierenden Artefakt benötigt werden. Die zugehörige OCL Spezifikation für die Analyse von nicht benötigten `imports` ausgehend von `MAAModelFiles` ist in Listing 10.14 dargestellt.

```
association /unusedImports [*] MAAModelFile -> Artifact [*];
context MAAModelFile inv:
  unusedImports == imports.removeAll(typeUsage);
```
CD «AM4A»

Listing 10.14: Analyse von ungenutzten Beziehungen für `MAAModelFiles`.

Um die Analysen auch auf andere MAA-Projekte übertragen zu können, müssen auch die in Abschnitt 6.6.4 vorgestellten Analysen zur Berechnung von ungenutzten `imports` für Java- bzw. CD-Artefakte durchgeführt werden.

Ein Auszug aus dem Ergebnis dieser Analyse ist in Listing 10.15 dargestellt. Das vollständige Analyse-OD lässt sich in Anhang G.2 finden.

```
link unusedImports @BumperBot!maa -> @Datatypes!cd(!2);
link unusedImports @BumperBot!maa -> @Datatypes!cd(!4);
link unusedImports @BumperBot!maa(!2) -> @Datatypes!cd(!2);
link unusedImports @BumperBot!maa(!2) -> @Datatypes!cd(!4);
```
OD «ArtData»

Listing 10.15: Ungenutzte `imports`-Links der BumperBot-Eingabeartefakte.

Es wurden insgesamt vier ungenutzte `imports`-Beziehungen erkannt. Dabei sind jeweils zwei ungenutzte `imports`-Beziehungen pro Variante des BumperBots vorhanden. Diese *Modell Smells* [ABT10] sollten im Zuge der weiteren Entwicklungen des BumperBots entfernt werden.

Die OCL-Spezifikation der zweiten Analyse zur Identifizierung ungenutzter Artefakte ausgehend von den Hauptartefakten der beiden Varianten ist in Listing 10.16 dargestellt.

```
context Artifact a, MAAModelFile m inv:
  (m.simpleName == "BumperBot") implies (a in m.refersTo**);
```
CD «AM4A»

Listing 10.16: Analyse von ungenutzten Artefakten für die BumperBot-Applikation.

Die beiden Hauptartefakte lassen sich hier über ihren Kurznamen (`simpleName`) identifizieren. Bei der Analyse werden transitiv alle ausgehenden `refersTo`-Beziehungen betrachtet.

Das Ergebnis der Durchführung der Analyse muss somit für jede Variante separat betrachtet werden. Beide Varianten nutzen kein Artefakt, welches sich der jeweils anderen Variante zuordnen lässt. Die Menge der übrigen ungenutzten Artefakte stimmt für beide Varianten überein. Diese Menge besteht aus neun ungenutzten Java-Platzhaltern der Bibliothek. Diese Artefakte können somit entfernt werden, falls ausgeschlossen werden kann, dass sie im Zuge der Weiterentwicklungen doch noch genutzt werden.

Die Zahlen zu der durchgeführten Analyse sind in Tabelle 10.17 zusammengefasst.

	A	B	Bib. (Java)	Bib. (MAA)	$\sum$
Artefakte	2	4	11	7	24
Ungenutzte Artefakte	-	-	9	-	9
	imports	**typeUsage**	**unusedImports**	**refersTo**	
Links	24	30	4	36	

Tabelle 10.17.: Die Ergebnisse der BumperBot-Analysen in Zahlen.

10.3. Make-basierte Buildprozesse

Ein weiterer Anwendungsfall für die artefaktbasierte Analyse ist das Beobachten von Make-basierten Buildprozessen. Um zu verstehen wie ein solcher Buildprozess abgelaufen ist, können bei der Ausführung der Makefiles Artefaktdaten erhoben werden. In diesem Abschnitt wird ein systematischer Ansatz zur Instrumentierung von Makefiles vorgestellt, welcher die Protokollierung von Artefaktdaten während der Ausführung des Buildprozesses ermöglicht. Mit Hilfe dieses Ansatzes ist sogar eine automatisierte Instrumentierung von Makefiles möglich. Abbildung 10.18 gibt einen Überblick über den Ansatz.

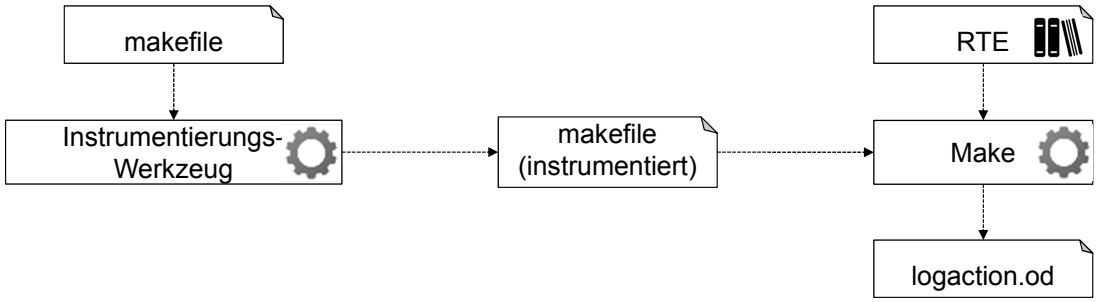

Abbildung 10.18.: Überblick über den Ansatz zur automatisierten Instrumentierung von Makefiles und dem Beobachten Make-basierter Buildprozesse.

Ein Werkzeug in Form eines Shell-Skripts liest dabei ein Makefile ein und produziert ein instrumentalisiertes Makefile als Ausgabe. Dieses Makefile kann wie das ursprüngliche Makefile genutzt werden. Der Unterschied zwischen den Makefiles besteht darin, dass bei dem instrumentierten Makefile die Ausführung von Regeln mitprotokolliert wird. Zu diesem Zweck wird eine RTE ebenfalls in Form von Shell-Skripten genutzt. Da Aktionen zur Laufzeit mitprotokolliert werden, handelt es sich bei diesem Anwendungsfall um eine dynamische Datenextraktion (siehe Abschnitt 3.3.2). Die Artefaktdaten, die hier erhoben werden, passen zu einer erweiterten Variante des in Abschnitt 6.5.2 dargestellten Teils des Artefaktmodells. Die Erweiterungen sind in Listing 10.19 dargestellt.

Es wurde eine Aktion für die Ausführung einer Regel und eine Aktion für das Löschen von Artefakten hinzugefügt. Zusätzlich wurden Assoziationen für gelöschte Artefakte und

CD «AM»

```
1  class RuleExecution extends Action {
2    String ruleName;
3  }
4
5  class ArtifactDelete extends Action {
6  }
7
8  association deletes [1] ArtifactDelete -> Artifact [*];
9
10 association target [1] RuleExecution -> Artifact [0..1];
11 association input [*] RuleExecution -> Artifact [*];
12
13 association /output [1] RuleExecution -> Artifact [*];
14 association /additionalResults
15   [1] RuleExecution -> Artifact [*];
16 association /deleted [1] RuleExecution -> Artifact [*];
17
18 context RuleExecution inv:
19   output == {a in ArtifactCreate | a in subaction**}.creates;
20
21 context RuleExecution inv:
22   additionalResults == output.remove(target);
23
24 context RuleExecution inv:
25   deleted == {a in ArtifactDelete | a in subaction**}.deletes;
26 }
```

Listing 10.19: Erweiterung des Artefaktmodells für das Monitoring Make-basierter Buildprozesse.

die Eingaben sowie Ausgabe der Regel hinzugefügt, mit deren Hilfe sich weitere abgeleitete Assoziationen berechnen lassen. Make-Regeln haben jeweils ein Ziel sowie mehrere Make-Abhängigkeiten. Sowohl das Ziel als auch die Abhängigkeiten werden häufig durch Artefakte repräsentiert. Es ist aber auch möglich, dass das Ziel sowie Make-Abhängigkeiten nicht durch Artefakte repräsentiert sind. In diesen Fällen wir das Ziel als Regelname angesehen und Make-Abhängigkeiten bestehen dann direkt zu diesen Regeln. Neben dem Zielartefakt können bei der Regelausführung weitere Artefakte produziert werden. Alle produzierten Artefakte sind in der `output`-Assoziation zusammengefasst während die `additionalResults`-Assoziation nur die zusätzlich produzierten Artefakte enthält.

In diesem Abschnitt werden keine konkreten Analysen durchgeführt, sondern stattdessen eine Methodik und ein zugehöriges Instrumentierungswerkzeug zur Protokollierung von Artefaktdaten für Make-basierten Buildprozesse vorgestellt. Die so gewonnenen Daten können zur Analyse komplexer Make-basierter Buildprozesse genutzt werden. Durch die Verwendung von

MontiArT ist dabei auch eine kombinierte Analyse dieser Daten mit den Daten eingebundener MDD-Werkzeuge möglich.

In den folgenden beiden Abschnitten wird zunächst die Methodik und anschließend das zugehörige Werkzeug zur automatischen Instrumentierung der Makefiles vorgestellt.

10.3.1. Methodik zur Instrumentierung von Makefiles

Wie in den Listings 10.20 und 10.21 beispielhaft dargestellt, wird bei der notwendigen Instrumentierung des Makefiles vor und nach jeder Regel die `startrule`- bzw. `endrule`-Funktion der RTE aufgerufen.

Make
```
1 $(TARGET): 1.txt 2.txt
2   @cat 1.txt > $(TARGET)
3   @cat 2.txt >> $(TARGET)
```

Listing 10.20: Beispiel für eine Make-Regel.

Make
```
1 $(TARGET): 1.txt 2.txt
2   @sh startrule.sh $@ $^
3   @cat 1.txt > $(TARGET)
4   @cat 2.txt >> $(TARGET)
5   @sh endrule.sh $@
```

Listing 10.21: Angepasste Make-Regel nach der Instrumentierung.

Als Parameter wird der `startrule`-Funktion sowohl das Ziel der Regel (`$@`) als auch die Make-Abhängigkeiten zu anderen Regeln und Eingabeartefakten (`$^`) übergeben. Die `startrule`-Funktion protokolliert nun zusätzlich alle relevante Dateien und das Datum ihrer letzten Änderung. Zusätzlich werden das Startdatum der Regelausführung und der Regelname, welcher sich aus den beiden übergebenen Parametern berechnen lässt, protokolliert. Existieren in den protokollierten Daten ein Artefakt für das Regelziel, wird ein entsprechender Link zum Zielartefakt (`target`) hinzugefügt. Analog dazu wir für alle Make-Abhängigkeiten der Regel geprüft, ob es sich jeweils um eine Abhängigkeit zu einem Artefakt handelt. Ist dies der Fall, so wird ein Link zu dem Eingabeartefakt der Regel (`input`) hinzugefügt.

Die `endrule`-Funktion fügt der `RuleExecution` den aktuellen Zeitpunkt als Enddatum hinzu. Darüber hinaus wird der aktuelle Stand der relevanten Dateien mit dem protokollierten Stand zu Beginn der Regelausführung verglichen, wodurch sich `ArtifactCreate`- und `ArtifactDelete`-Aktionen als Unteraktionen (`subaction`, siehe Abschnitt 6.5.1) der `RuleExecution` erstellen lassen. Die Modifikation existierender Dateien wurde gemäß der Überlegungen aus Abschnitt 2.2 ausgeschlossen, weshalb sie an dieser Stelle ebenfalls

nicht betrachtet wird. Das Erheben von Start- und Endzeitpunkt von Unteraktionen ist durch diesen Ansatz allerdings nicht möglich. Auch das Protokollieren von `ArtifactRead`- und `ArtifactExistenceCheck`-Aktionen ist nicht ohne zusätzliche Instrumentierung möglich. Es bleibt außerdem zu erwähnen, dass sich durch Instrumentierung von Programmcode dessen Ausführungsdauer verlängert [Lig09], was hier bei der Betrachtung der Start- und Endzeitpunkte berücksichtigt werden sollte.

Alternativ zu der vorgestellten Vorgehensweise könnte man pro `ArtifactCreate`- und `ArtifactDelete`-Aktion eigene Start- und Endfunktionen in der RTE anbieten. Dadurch wäre es dann zusätzlich möglich, den Start- bzw. Endzeitpunkt dieser Unteraktionen zu protokollieren. Der Nachteil ist hierbei ein erhöhter Instrumentierungsaufwand durch die erhöhte Anzahl an Funktionsaufrufen. Darüber hinaus wird ein systematisches Vorgehen bei der Instrumentierung erschwert, da nicht nur an den zwei fixierten Stellen pro Regel Funktionsaufrufe hinzugefügt werden sondern pro relevantem Statement, eine oder mehrere der genannten Funktionen aufgerufen werden. Darüber hinaus ist es bei Make üblich den Rumpf einer Make-Regel als ein einzelnes Make-Kommandos zu verstehen, auch wenn dort mehrere Shell-Kommando verwendet werden. Aus den genannten Gründen wurde diese Alternative nicht gewählt.

10.3.2. Werkzeug zur automatisierten Instrumentierung von Makefiles

Der im vorherigen Abschnitt beschriebene systematische Ansatz zur Instrumentierung von Makefiles lässt sich durch ein einfaches Werkzeug automatisieren. Das Werkzeug liest wie in Abbildung 10.18 dargestellt ein Makefile ein und erstellt als Ausgabe ein instrumentiertes Makefile. Listing 10.22 zeigt den entsprechenden Quellcode in Form eines Shell-Skripts.

SH

```
cat $1 | awk '
  BEGIN {inTab=0}
  /^\t/ && inTab==0 {
    print "\t@sh startrule.sh $@ $^";
    inTab=1;
  }
  !/^\t/ && inTab==1 {
    print "\t@sh endrule.sh $@";
    inTab=0;
  }
  {print $0}
' > $2
```

Listing 10.22: Werkzeug zur automatisierten Instrumentierung von Makefiles.

Das Werkzeug bekommt beim Aufruf das zu instrumentierende Makefile als ersten Parameter übergeben. Der Inhalt wird per *awk*-Programm manipuliert. Das manipulierte Ergebnis wird

dann in die Datei geschrieben, die als zweiter Parameter dem Werkzeug übergeben wurde. Ein Makefile mit einem anderen Namen als „makefile“ lässt sich mit dem Parameter `-f`, wie in Listing 10.23 dargestellt, aufrufen.

```
make -f makeFileWithCustomName [Targets]
```

Listing 10.23: Aufruf von eines Makefile mit beliebigem Namen.

Das awk-Programm speichert, ob die letzte Zeile mit einem Tab begonnen hat (`inTab`). Wenn eine Zeile mit einem Tab beginnt und dies bei der letzte Zeile nicht der Fall war, so handelt es sich um die erste Anweisung einer Regel. An dieser Stelle wird nun der Aufruf der `startrule`-Funktion mit den entsprechenden Parametern eingefügt. Wenn die letzte Zeile mit einem Tab begonnen hat, die aktuelle Zeile aber nicht, ist das Ende der Regel erreicht. Hier wird nun der Aufruf der `endrule`-Funktion eingefügt. Die beiden Funktionen wurden in diesem Beispiel ebenfalls durch Shell-Skripte der RTE (siehe Abbildung 10.18) realisiert.

Um das vorgestellte Werkzeug einsetzen zu können, müssen die folgenden Bedingungen für Make-Regeln eingehalten werden:

- Nach jeder Make-Regel muss eine Leerzeile stehen.
- Jede Zeile im Rumpf einer Make-Regel (*recipe* [SMS16]) muss mit einem Tab beginnen.

Sollte eine der Bedingungen nicht eingehalten werden können, so muss das Werkzeug entsprechend angepasst oder die Instrumentierung von Hand durchgeführt werden.

10.4. Inkrementelle Generierung mit MontiCore

Ein weiterer und sehr wichtiger Anwendungsfall für die artefaktbasierte Analyse ist die inkrementelle Generierung mit MontiCore, die in diesem Abschnitt vorgestellt wird. Bei der inkrementellen Generierung werden zuvor erhobene Artefaktdaten zur effizienten Ausführung von Buildprozessen genutzt, die MontiCore-Aufrufe enthalten. Zur Definition des Buildprozesses wird in diesem Abschnitt Make verwendet. Die inkrementelle Generierung ist dabei wie folgt definiert:

Definition 10.4.1 (Inkrementelle Generierung)**.** *In einem inkrementellen Buildprozess, der aus mehreren Generierungsschritten besteht, werden nur genau die Prozessschritte ausgeführt, bei denen sich relevante Eingabeartefakte seit der letzten Ausführung geändert haben oder Ausgabeartefakte fehlen.*

Nach dieser Definition werden nicht nur Generierungsschritte, sondern alle Schritte des Buildprozesses inkrementell ausgeführt. Würden nicht alle Schritte ausschließlich bedarfsorientiert ausgeführt, könnten die vorhandenen Prozessabhängigkeiten dazu führen, dass große

Teile des Buildprozesses unnötig neu ausgeführt werden, was besonders bei einer mehrstufigen Generierung lange Ausführungszeiten zur Folge haben kann und somit das von agilen Entwicklungsmethoden benötigte schnelle Feedback [Bec04] verhindert.

Bei der Entwicklung von MontiCore wird Maven zur Buildprozessautomatisierung eingesetzt. Während sich dadurch die einzelnen Komponenten von MontiCore gut modularisieren und sich Beziehungen zwischen den einzelnen Maven-Modulen komfortabel verwalten lassen, gibt es keine Möglichkeit, eine inkrementelle Generierung direkt mit Maven zu realisieren. Dies liegt daran, dass Maven phasenbasiert ausgeführt wird und kein übergeordnetes Wissen über Ein- und Ausgaben der Generierungsbeziehungen von Artefakten hat. Die einzige Möglichkeit ist deshalb, die eingebundenen Werkzeuge jeweils selbst entscheiden zu lassen, ob sie aufgrund von Artefaktänderungen ausgeführt werden müssen oder nicht. Diese Aufgabe kann auch in Maven-Plugins ausgelagert werden [Sou16]. Trotzdem muss dann für jedes Plugin einzeln geprüft werden, ob eine Ausführung notwendig ist. Ein häufiges Problem ist dabei, dass Entwickler bei einfachen und schnell auszuführenden Befehlen, wie beispielsweise Kopierbefehlen, auf diese Prüfung verzichten, da sich die Ausführungszeit des einzelnen Plugins dadurch nicht wesentlich verbessern lässt. Dies führt jedoch dazu, dass auch alle nachfolgenden Schritte in einem eigentlich inkrementellen Generierungsprozess jedes Mal ausgeführt werden.

Im Gegensatz zu Maven bietet Make die Möglichkeit, die benötigten Eingaben für einzelne Prozessschritte direkt anzugeben. Dies versetzt Make in die Lage, die für gegebene Artefaktänderungen notwendigen Prozessschritte zu identifizieren und dann nur diese Schritte auszuführen. Somit ist Make für die inkrementelle Generierung deutlich besser geeignet und wird in diesem Abschnitt zu deren Realisierung genutzt.

Der in diesem Kapitel vorgestellte Ansatz basiert auf Abschnitt 6.6.5. Dort wurde bereits spezifiziert, dass `ToolExecutions` erneut ausgeführt werden müssen, wenn sich deren Eingaben ändern oder entfernt werden und dass auch alle nachfolgenden Werkzeugausführungen erneut ausgeführt werden müssen. Das Hinzufügen von Eingabeartefakten konnte dort allerdings nicht berücksichtigt werden. Eine Neuausführung von MontiCore ist auch in solchen Fällen notwendig. Neben den Eingabeartefakten, die direkt bei Aufruf von MontiCore übergeben werden, prüft MontiCore während der Ausführung die Existenz von *handgeschriebenen Quellcodedateien* und reagiert entsprechend darauf [HR17]. Durch das Anlegen solcher Dateien ermöglicht MontiCore dem Generatornutzer die direkte Integration von handgeschriebenem und generiertem Code, wodurch sich sowohl die Attribute als auch die Signatur der generierten Klassen ergänzen lassen ohne diese manuell anpassen zu müssen. Eine Änderung eines solchen handgeschrieben Artefakts soll nicht zu einer Neuausführung führen, da der Inhalt dieser Artefakte keinen Einfluss auf den Generierungsprozess von MontiCore hat. MontiCore prüft bei Ausführung nur die Existenz dieser Dateien, verarbeitet deren Inhalt aber nicht [HR17]. Die zugehörige `existenceImpact`-Beziehung zwischen den geprüften und den erstellten Artefakten wurde bereits in Abschnitt 6.5.3 eingeführt.

Der handgeschriebene Code kann MontiCore bei Aufruf in Form von einem oder mehreren Quellcodeordnern übergeben werden. Innerhalb dieser Ordner müssen Dateien entsprechend ihrer Paketstruktur organisiert sein (vgl. Abschnitt 6.1.2). Dabei müssen jedoch nicht alle

Dateien des Ordners als handgeschriebene Quellcodedateien der oben beschriebenen Form dienen. Es können auch weitere Dateien enthalten sein, die beispielsweise ausschließlich als Eingabe für einen der nachfolgenden Schritte dienen. Solche Dateien werden im Folgenden als *nicht relevante handgeschriebene Codedateien* bezeichnet, wobei dies im oben beschriebenen Zusammenhang zu sehen ist. Dies führt zu folgenden Anforderungen für die Ausführung von MontiCore:

- Eine erneute Ausführung von MontiCore ist notwendig, wenn
  1. ein Eingabeartefakt hinzugefügt oder geändert wurde.
  2. eine relevante handgeschriebenen Codedatei hinzugefügt wurde.
  3. eine relevante handgeschriebene Codedatei entfernt wurde.
- Eine erneute Ausführung von MontiCore wird nicht ausgelöst, durch
  4. Änderungen an relevanten handgeschriebenen Codedateien.
  5. Hinzufügen, Entfernen von oder Änderungen an nicht relevanten handgeschriebenen Codedateien.

Das Entfernen oder Ändern von Ausgaben wird wegen der Vorgabe aus Abschnitt 2.2, dass generierte Artefakte nicht manuell verändert werden dürfen, nicht weiter betrachtet. MontiCore besitzt bereits einen Mechanismus, mit dem bei jeder Ausführung nach diesen Anforderungen entschieden wird, ob die Generierung durchgeführt werden muss. Hierzu muss MontiCore jedoch jedes Mal aufgerufen werden, was viel Zeit kostet. Nach Definition 10.4.1 kommt dieser Mechanismus nicht für einen vollständigen inkrementellen Buildprozess infrage, da ein solcher unnötiger Aufruf gar nicht erst stattfinden darf.

Abbildung 10.24 zeigt die Artefaktbeziehungen, die in einem beispielhaften Generierungsprozess, der inkrementell ausgeführt werden soll, vorliegen. An zwei Stellen sind `existenceImpact`-Beziehungen (vgl. Abschnitt 6.5.3) von handgeschriebene Codedateien (`hwc1` und `hwc2`) zu den von MontiCore erstellten Artefakten vorhanden.

Die linke Seite der Abbildung beschreibt die Generierung und Ausführung des modellverarbeitenden Werkzeugs `automaton-tool.jar`. Analog zu dem Beispiel aus [MCG18] liest dieses Werkzeug ein Automatenmodell (`PingPongTF.aut`) ein, prüft das Modell auf bestimmte Eigenschaften und gibt einen Bericht über die Prüfung des Modells aus (`Result.txt`). Zusätzlich zu dem Beispiel aus [MCG18] wird auf der rechten Seite von Abbildung 10.24 ein Werkzeug zur Modelltransformation (`trafo-tool.jar`) erstellt und ausgeführt, welches das Eingabemodell `PingPong.aut` anhand des Transformationsmodells `RenameState.tf` transformiert. Da das Transformationswerkzeug ebenfalls Automaten verarbeitet, werden zu dessen Kompilierung die zugehörigen Typen aus `automaton-tool.jar` benötigt. In dem dargestellten Beispiel werden die Werkzeuge nicht als Fat-Jars [FJE17] erstellt, weshalb auch zur Ausführung der Transformation, also zur Erstellung von `PingPongTF.aut` die Typen aus `AutomatonTool.jar` benötigt werden. Listing 10.25 zeigt das initiale Makefile, das zur inkrementellen Generierung genutzt werden soll.

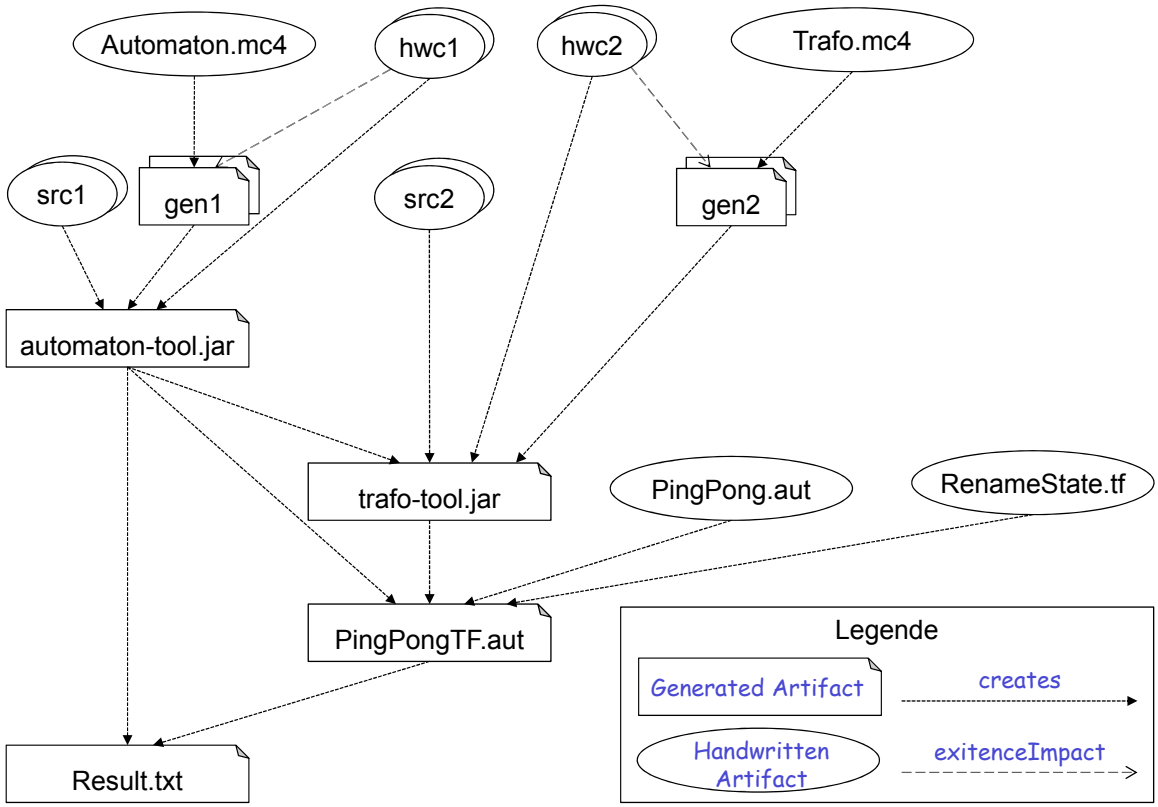

Abbildung 10.24.: Artefaktbeziehungen eines beispielhaften, inkrementellen Generierungsprozesses mit MontiCore.

Das Makefile besteht aus sechs Regeln, welche die zu Abbildung 10.24 zugehörigen Ausführungsschritte realisieren. Wenn ein Schritt einen Ordner als Eingabe bekommt, wurde in der zugehörigen Make-Regel eine Make-Abhängigkeit auf alle in dem Ordner enthaltenen Dateien spezifiziert, wie es zum Beispiel für die Regel `gen` der Fall ist. Eine Ausnahme bilden hier die beiden `gen`-Ordner. Da diese erst bei Ausführung des Prozesses erstellt werden, können sie direkt als Make-Abhängigkeit genutzt werden. Die Regel `all` hat nur eine Make-Abhängigkeit zu der letzten auszuführenden Regel (`Result.txt`) des Buildprozesses. Make identifiziert bei Aufruf dann die Schritte des Gesamtprozesses, für die sich Eingaben geändert haben und führt dann nur genau diese Schritte aus. Ändert sich bei Ausführung einer Regel eine Datei, zu der eine weitere Regel eine Make-Abhängigkeit hat, so wird diese Regel ebenfalls ausgeführt. Dies kann somit auch zur vollständigen Ausführungen des Buildprozesses führen.

Bei dem in Listing 10.25 dargestellten Makefile ist eine korrekte Reaktion auf Hinzufügen, Änderung und Entfernen von handgeschriebenen Quellcodedateien nicht möglich. In den Fällen 1, 3 und 4 der oben beschriebenen Anforderungen wird die entsprechende `gen` Regel jeweils ausgeführt, MontiCore prüft dann allerdings bei der Ausführung der Regel selbst, ob neu

Make

```
src1files=$(shell find src1 -name "*.java" -type f)
hwc1files=$(shell find hwc1 -name "*.java" -type f)
src2files=$(shell find src2 -name "*.java" -type f)
hwc2files=$(shell find hwc2 -name "*.java" -type f)

all: Result.txt

gen1: grammars/Automaton.mc4 $(hwc1files)
  java -jar monticore-cli.jar grammars/Automaton.mc4 -out $@ \
    -hcp hwc1 -cl logging1.xml

automaton-tool.jar: $(src1files) $(hwc1files) gen1
  rm -rf build1
  mkdir -p build1
  javac -cp monticore-cli.jar -d build1 \
    -sourcepath "src1;gen1;hwc1" src1/automaton/AutomatonTool.java
  jar -cf $@ -C build1 .

gen2: grammars/Trafo.mc4 $(hwc2files)
  java -jar monticore-cli.jar grammars/Trafo.mc4 -out $@ \
    -hcp hwc2 -cl logging2.xml

trafo-tool.jar: $(src2files) $(hwc2files) gen2 automaton-tool.jar
  rm -rf build2
  mkdir -p build2
  javac -cp "monticore-cli.jar;automaton-tool.jar" -d build2 \
    -sourcepath "src2;gen2;hwc2" src2/trafo/TrafoTool.java
  jar -cf $@ -C build2 .

PingPongTF.aut: trafo-tool.jar automaton-tool.jar \
  models/PingPong.aut models/RenameState.tf
  java -cp "trafo-tool.jar;automaton-tool.jar;monticore-cli.jar" \
    trafo.TrafoTool models/PingPong.aut models/RenameState.tf $@

Result.txt: automaton-tool.jar PingPongTF.aut
  java -cp "automaton-tool.jar;monticore-cli.jar" \
    automaton.AutomatonTool PingPongTF.aut > $@
```

Listing 10.25: Initiales Makefile für die inkrementelle Generierung mit MontiCore.

generiert werden muss[2]. Die dabei erstellte Log-Datei muss dazu in einen anderen Ordner

[2]Hierbei wird ausgenutzt das GNU Make [SMS16] vor jeder Regelausführung dynamisch prüft, ob sich eine der Eingaben geändert hat. Somit führt hier eine Ausführung der Vorgängerregel nicht zwingend zu einer Ausführung der Nachfolgerregel.

(anstelle des `gen`-Ordners) gespeichert werden, da die Make-Abhängigkeit der anderen Regel zu dem entsprechenden `gen`-Ordner sonst zu einer Ausführung weiterer Prozessschritte führt. Somit wird zwar die Regel ausgeführt, was Definition 10.4.1 widerspricht, trotzdem wird keine erneute Generierung durchgeführt, weshalb dies als eingeschränkte inkrementelle Generierung angesehen werden kann. Fall 2 der oben aufgeführten Anforderungen kann hingegen auch nicht eingeschränkt durch das Makefile aus Listing 10.25 erfüllt werden. Bei Entfernen einer relevanten Codedatei wird die entsprechende Make-Regel nicht erneut ausgeführt, da in diesem Fall die Make-Abhängigkeit zur entfernten handgeschriebenen Codedatei auch entfernt wird. In Abschnitt 10.4.1 werden zwei verschiedene Ansätze zur Lösung dieses Problems vorgestellt.

10.4.1. Lösungsansätze

Make

```
.PHONY: force

.hwc1files: force
  @echo $(hwc1files) | cmp - $@ || echo $(hwc1files) > $@

.hwc2files: force
  @echo $(hwc2files) | cmp - $@ || echo $(hwc2files) > $@

gen1: grammars/Automaton.mc4 .hwc1files
  java -jar monticore-cli.jar grammars/Automaton.mc4 -out $@ \
    -hcp hwc1 -cl logging1.xml

gen2: grammars/Trafo.mc4 .hwc2files
  java -jar monticore-cli.jar grammars/Trafo.mc4 -out $@ \
    -hcp hwc2 -cl logging2.xml
```

Listing 10.26: Anpassungen Gegenüber dem Makefile aus Listing 10.25 für eine korrekte inkrementelle Generierung mit MontiCore.

Um auch Fall 2 der Anforderungen erfüllen zu können, kann das Makefile wie in Listing 10.26 angepasst werden. In der Abbildung wurden nur die Änderungen gegenüber Listing 10.25 dargestellt. Alle nicht dargestellten Bestandteile wurden unverändert übernommen. In dieser und der noch folgenden Anpassung in Listing 10.25 werden folgende Befehle und Operatoren verwendet

- `||`: Führt die rechte Seite nur dann aus, wenn die linke Seite *false* ist.
- `&&`: Führt die rechte Seite nur dann aus, wenn die linke Seite *true* ist.
- `cmp`: Vergleicht zwei Dateien. Die Option `-` wird verwendet werden, um statt einer Datei den Text der Standardeingabe einzulesen.

- `[ -e ]`: Prüft die Existenz einer Datei.
- `echo`: Gibt den übergebenen String auf der Standardausgabe aus.
- `touch`: Aktualisiert den Zeitstempel der Datei oder erstellt eine leere Datei, falls diese zuvor nicht existiert.

Die Regeln `gen1` und `gen2` haben jetzt eine Make-Abhängigkeit zu den Statusdateien `.hwc1files` und `.hwc2files` anstelle der handgeschriebenen Codedateien. Diese Statusdateien speichern die handgeschriebenen Codedateien der jeweils letzten Ausführung des Buildprozesses ab. Bei erneutem ausführen des Buildprozesses wird dann durch die neu hinzugefügten Make-Regeln geprüft, ob sich die aktuell vorhandenen handgeschriebenen Codedateien eines MontiCore-Aufrufs von denen in der entsprechenden Statusdatei unterscheiden. Nur bei einem Unterschied wird der Inhalt der Datei mit einer Liste der aktuellen handgeschriebenen Codedateien aktualisiert. Dies führt auch zu einer Aktualisierung des Zeitstempels, was wiederum zur Ausführung der abhängigen Make-Regeln führt. Die Regeln zur Berechnung der Statusdateien müssen bei jedem Durchlauf ausgeführt werden. Dies wird durch Angabe der `PHONY`-Regel und der zugehörigen Make-Abhängigkeit auf `force` realisiert. Dasselbe Verfahren lässt sich auch für die anderen Make-Regeln aus Listing 10.25 anwenden, die ebenfalls von Codedateien aus einem der Ordner abhängen. Somit führt das Entfernen von Eingabequellcodedateien auch dort zu einer erneuten Ausführung der Regeln. Die bereits vorhandenen Make-Abhängigkeiten zu den einzelnen Dateien werden hier allerdings nicht entfernt, damit eine Neuausführung der Regel auch weiterhin bei Änderungen an diesen Codedateien stattfindet. Listing 10.27 zeigt dies exemplarisch für die Regel `automaton-tool.jar`.

Make

```
.src1files: force
  @echo $(src1files) | cmp - $@ || echo $(src1files) > $@

automaton-tool.jar: $(src1files) $(hwc1files) .src1files \
  .hwc1files gen1
  rm -rf build1
  mkdir -p build1
  javac -cp monticore-cli.jar -d build1 \
    -sourcepath "src1;gen1;hwc1" src1/automaton/AutomatonTool.java
  jar -cf $@ -C build1 .
```

Listing 10.27: Anpassung der Regel `automaton-tool.jar`, um auf das Entfernen von Eingabedateien reagieren zu können.

Die in Listing 10.26 gezeigten Änderungen führen dazu, dass die Make-Regeln mit den MontiCore Aufrufen für die Fälle 1, 2 und 3 der Anforderungen korrekt reagieren, da nur auf Hinzufügen oder Entfernen, nicht aber auf Änderungen an relevanten handgeschriebenen Codedateien reagiert wird. Fall 4 führt hingegen weiterhin zu einer erneuten Ausführung

der Regeln, da auch auf Hinzufügen und Entfernen von nicht relevanten handgeschriebenen Codedateien reagiert wird. In diesen Fällen muss MontiCore weiterhin selbst prüfen, ob eine Neugenerierung notwendig ist. Somit konnte durch die Änderung aus Listing 10.26 ein eingeschränkter inkrementeller Generierungsprozess definiert werden.

Durch den beschriebenen Ansatz können somit zwar die Anforderungen in Bezug auf die Neugenerierung erfüllt werden, allerdings muss die Entscheidung, ob eine Neugenerierung notwendig ist, in manchen Fällen durch MontiCore selbst getroffen werden, was Definition 10.4.1 widerspricht. Um einen vollständigen inkrementellen Buildprozess zu erhalten und somit die Performance bei der Ausführungen zu steigern, muss verhindert werden, dass die entsprechende Make-Regel im vierten Fall ausgeführt wird. Wenn dies sichergestellt ist, kann die MontiCore interne Überprüfung abgeschaltet werden und die Entscheidung zur Neuausführung vollständig von Make getroffen werden.

Zur Lösung dieses Problems werden bei der Ausführung von MontiCore Skriptdateien basierend auf den Existenzprüfungen von handgeschriebenen Codedateien (`ArtifactExistenceChecks`) generiert, die dann bei der nächsten Ausführung des Buildprozesses die Entscheidung, ob die Voraussetzungen für eine erneute Ausführung der Regel in Bezug auf den handgeschriebenen Code erfüllt sind, treffen. Diese Erweiterung von MontiCore um einen zusätzlichen Reporter (vgl. Abschnitt 9.2.3) wurde im Rahmen dieser Arbeit vorgenommen.

Um einen vollständig inkrementellen Buildprozess zu erhalten, muss anstelle der zuvor gezeigten Erweiterung (Listing 10.26) das Makefile aus Listing 10.25 entsprechend der Erweiterungen aus Listing 10.28 angepasst werden. Alle nicht dargestellten Regeln wurden unverändert aus Listing 10.25 übernommen.

Für die beiden MontiCore-Ausführungen wurde die Make-Abhängigkeit auf die handgeschriebenen Quellcodedateien entfernt und durch Make-Abhängigkeiten auf die Statusdateien `.igf1` bzw. `.igf2` ersetzt. Diese leeren Dateien werden durch die generierten Skripte (IncGenCheck.sh) erstellt und deren Zeitstempel genau dann erneuert, wenn eine erneute Ausführung von MontiCore aufgrund von relevanten Änderungen von handgeschriebenem Code notwendig ist. Eine Aktualisierung des Zeitstempels von `.igf1` führt hier zu einer erneuten Ausführung von `gen1`, während eine Aktualisierung des Zeitstempels von `.igf2` zu einer Ausführung von `gen2` führt. Durch die vorgenommenen Änderungen wird MontiCore somit nur noch genau dann aufgerufen, wenn auch neu generiert werden muss. Somit wird die Option zum verändern des Ablageorts der Log-Datei (`-cl`) nicht mehr zwingend benötigt und die interne Prüfung von MontiCore kann übersprungen werden (`-f`).

Bei der ersten Ausführung des Buildprozesses existieren noch keine Statusdateien, weshalb in diesem Fall die Statusdateien initial angelegt werden (Zeilen 7 und 11). In allen anderen Fällen ist der Buildprozess bereits ausgeführt worden, weshalb auf die zuvor generierten `IncGenCheck.sh`-Dateien zugegriffen werden kann. Diesen Skripten wird bei Aufruf die jeweilige Statusdatei als Parameter übergeben, deren Zeitstempel erneuert wird, falls die Notwendigkeit für eine erneute Ausführung von dem jeweiligen Skript erkannt wird. Die generierten Skripte sind somit für die effiziente Überprüfung von Änderungen des handgeschriebenen

```
igf1Check=gen1/reports/IncGenCheck.sh                                    Make
igf2Check=gen2/reports/IncGenCheck.sh

.PHONY: force

.igf1: force
  @[ -e .igf1 ] || touch .igf1
  @[ -e $(igf1Check) ] && sh $(igf1Check) .igf1 || true

.igf2: force
  @[ -e .igf2 ] || touch .igf2
  @[ -e $(igf2Check) ] && sh $(igf2Check) .igf2 || true

gen1: grammars/Automaton.mc4 .igf1
  java -jar monticore-cli.jar grammars/Automaton.mc4 -out $@ \
    -hcp hwc1 -f

gen2: grammars/Trafo.mc4 .igf2
  java -jar monticore-cli.jar grammars/Trafo.mc4 -out $@ \
    -hcp hwc2 -f
```

Listing 10.28: Anpassungen des Makefile für die effiziente inkrementelle Generierung mit MontiCore.

Quellcodes verantwortlich. Listing 10.29 zeigt einen Ausschnitt einer solchen generierten `IncGenCheck`-Datei.

```
[ -e C:/mc-workspace/hwc1/automaton/_ast/ASTState.java ] || \          SH
(touch $1; exit 0;)
# ...
[ -e C:/mc-workspace/hwc1/automaton/_ast/ASTTransition.java ] && \
(touch $1; exit 0;)
# ...
```

Listing 10.29: Ausschnitt eines generierten Shell-Skripts zur optimierten Prüfung auf relevante Änderungen von handgeschriebenem Quellcode.

Diese Skriptdateien führen ebenfalls `ArtifactExitenceChecks` durch und aktualisieren die Statusdateien genau dann, wenn sich das Ergebnis eines `ArtifactExistence-Checks` von dem Ergebnis derselben Prüfung beim vorherigen Durchlauf von MontiCore

unterscheidet (Fälle 1 und 2 der Anforderungen). Für jeden `ArtifactExistenceCheck` der letzten Ausführung von MontiCore wurde dazu eine einzelne Zeile im Skript generiert[3]

Die Zeilen werden alle nach demselben Schema generiert, unterscheiden sich aber in der Wahl des Operators. Je nachdem, ob bei der Generierung ein erfolgreicher oder ein erfolgloser `ArtifactExistenceCheck` durchgeführt wurde, wird ein anderer Operator verwendet. Erfolgreiche Prüfungen sind jene, bei denen das gesuchte Artefakt gefunden wurde, wohingegen das Fehlen eines gesuchten Artefakts zu einer erfolglosen Prüfung führt.

Für Fall 1 der Anforderungen war der `ArtifactExistenceCheck` der vorherigen Ausführung von MontiCore erfolglos, weshalb die Statusdatei genau dann aktualisiert werden muss, wenn der aktuelle `ArtifactExistenceCheck` erfolgreich ist. Dieses Verhalten wird durch Zeile 4 des generierten Skripts aus Listing 10.29 hervorgerufen, da die rechte Seite des Skriptes nur dann ausgeführt wird, wenn der aktuelle `ArtifactExistenceCheck` erfolgreich ist. Dieses Verhalten wird durch Verwendung des `&&`-Operators erzielt. Im Gegensatz dazu wird das Verhalten für Fall 2 durch die Verwendung des `||`-Operators hervorgerufen. Die rechte Seite wird hier nur dann ausgeführt, wenn der aktuelle `ArtifactExistenceCheck` erfolglos ist. Solche Zeilen werden für erfolgreiche `AritfactExistenceChecks` (der vorherigen Ausführung von MontiCore) erstellt. Ein Beispiel für eine solche Zeile ist in Listing 10.29 in Zeile 1 dargestellt.

Auf der rechten Seite der Operatoren wird jeweils eine Aktualisierung der Statusdatei per `touch`-Kommando vorgenommen. Bei der ersten Aktualisierung der Statusdatei wurde bereits eine relevante Änderung der handgeschriebenen Codedateien erkannt, weshalb die Ausführung des Skripts per `exit`-Kommando beendet wird. Wird das Skript bis zum Ende durchlaufen, so wurde kein Grund für eine Neugenerierung gefunden und der Zeitstempel der Statusdatei wird nicht aktualisiert. Somit muss für die Fälle 3 und 4 der Anforderungen kein zusätzlicher Skriptcode generiert werden.

Der gezeigte Lösungsansatz führt zu einem inkrementellen Generierungsprozess nach Definition 10.4.1, da alle Anforderungen erfüllt sind und die dazu notwendigen Entscheidungen getroffen werden können, ohne die `gen`-Regeln ausführen zu müssen. In Abschnitt 10.4.2 wird die Performance der beiden Lösungsansätze dieses Abschnitts verglichen.

10.4.2. Evaluation

Um zu überprüfen, wie groß der Performancegewinn durch die Optimierung unter Verwendung der generierten `IncGenSkript`-Dateien tatsächlich ist, wurde eine Messung durchgeführt. Als Testumgebung wurde ein i7-2640M @ 2.80GHz Prozessor, Windows 7, CygWin und eine SSD genutzt. Dabei wurde die Ausführungszeit für einen Aufruf der `gen1`-Regel (`make gen1`) für die beiden erweiterten Varianten aus den Abbildungen 10.26 und 10.28 gemessen. Bei dem zugehörigen MontiCore-Aufruf handelt es sich, wie oben beschrieben, um das Beispiel aus [MCG18], welches fünf relevante, handgeschriebene Codedateien enthält. Bei einer

[3] In Listing 10.29 wurden die rechten Seiten der Operatoren aus Platzgründen jeweils in einer eigenen Zeile dargestellt.

einzelnen Ausführung von MontiCore werden 40 Dateien generiert und somit ebenfalls 40 `ArtifactExistencChecks` durchgeführt. Zur Berechnung der in Tabelle 10.30 dargestellten Werte wurden je zehn Messungen durchgeführt.

	Ohne IncGenSkript			**Mit IncGenSkript**		
	Min	Max	Avg	Min	Max	Avg
Erstmalig	5385	6025	5694	5494	6009	5735
Ohne Änderung	312	343	331	265	281	275
Irrelevantes Hinzufügen	1625	1797	1722	265	281	275

Tabelle 10.30.: Ausführungsdauern (Minimum, Maximum, Durchschnitt) in ms für den Aufruf der `gen1`-Regel der verschiedenen Makefile-Varianten.

Bei den Messungen für *Erstmalig* wurde jeweils die erstmalige Ausführungsdauer des Buildprozesses gemessen. Hier unterschieden sich die Ausführungsdauern im Hinblick auf die Gesamtdauer nur geringfügig. Während die Abweichung zwischen den minimalen und maximalen Ausführungsdauern stärker variierten. Für beide Varianten musste MontiCore ausgeführt werden, da eine Neugenerierung notwendig ist. Unter Verwendung der `IncGenSkript`-Variante wird in diesem Fall die MontiCore interne Prüfung übersprungen. Da diese für den Fall der Neugenerierung jedoch sehr effizient durchgeführt wird und die Java-VM in beiden Fällen gestartet werden muss, ist hier kein nennenswerter Unterschied messbar.

Bei den Messungen für den Fall *Ohne Änderung* wurde der Buildprozess einmalig ausgeführt und anschließend keine der Eingaben verändert, sodass für beide Variante die Regel `gen1` erst gar nicht ausgeführt werden musste. Interessant ist an dieser Stelle, dass in allen Fällen die Berechnung der Statusdateien unter Verwendung der `IncGenSkript`-Datei schneller war als die Variante unter Verwendung des `cmp`-Kommandos. Trotzdem ist dieser Performancegewinn im Millisekunden Bereich und somit nicht ausschlaggebend für die Wahl dieser Variante.

Der große Vorteil der Variante unter Verwendung des `IncGenSkripts` und somit einem vollständig inkrementellen Buildprozess nach Definition 10.4.1 im Vergleich zu der ersten Variante wird bei der Messung von Fall *Irrelevantes Hinzufügen* sichtbar. Dazu wurde der Buildprozess einmalig ausgeführt und anschließend eine einzelne irrelevant handgeschriebene Codedatei hinzugefügt. Ausgehend von dieser Situation wurden die zehn Messungen ohne weitere Änderungen durchgeführt. Während für die Variante mit `IncGenSkript` die Regel in keinem Durchlauf ausgeführt wurde, wurde die Regel in der anderen Variante jedes Mal ausgeführt, obwohl eine Änderung nur einmalig vor der ersten Messung stattgefunden hat. Dies ist dadurch begründet, dass die Statusdatei einen neueren Zeitstempel als der `gen1`-Ordner hat und MontiCore bei jedem Durchlauf intern entscheidet, dass nicht erneut generiert werden muss. Somit wird auch der Zeitstempel des `gen1`-Ordners nicht aktualisiert, was bei der jeweils nächsten Ausführung des Buildprozesses auch zu einer wiederholten Ausführung der Make-Regel führt, auch wenn keine weiteren Änderungen vorgenommen wurden. Dies ist insofern ineffizient, als eine MontiCore interne Überprüfung im Gegensatz zu der skriptbasierten Variante eine spürbar schlechtere Ausführungszeit von durchschnittlich 1,4 Sekunden hat.

Somit konnte die Performance für die Überprüfung, ob aufgrund von handgeschriebenem Quellcode neu generiert werden muss, für Fall 4 der oben beschrieben Anforderungen deutlich gesteigert werden. Gerade wenn viele MontiCore-Aufrufe in den iterativen Generierungsprozess eingebunden werden, wird die Auswirkung spürbar. Ein zusätzlicher Nachteil der ersten Variante ist, dass nach einem einmaligen für MontiCore irrelevanten Hinzufügen oder Entfernen einer handgeschriebenen Codedatei die MontiCore interne Prüfung bei jedem Durchlauf des Buildprozesses durchgeführt werden muss, auch wenn keine weiteren Dateien hinzugefügt oder entfernt wurden.

Durch die in diesem Kapitel vorgestellte Vorgehensweise kann eine vollständige und effiziente inkrementelle Generierung mit MontiCore umgesetzt werden, sodass die Berechnung der Ausführungsreihenfolge weiterhin durch das Buildsystem durchgeführt wird und trotzdem die speziellen Anforderungen für eine Neuausführung von MontiCore berücksichtigt werden.

10.5. Artefakte bei der Testfallgenerierung im Automotivekontext

Im Zuge eines Industrieprojekts im Automotivekontext wurde ein Generator entwickelt, welcher aus Modellen der funktionalen Spezifikation automatisch ausführbare Testfälle generiert [KMS+17, Mil17, KMS+18]. Funktionsmodelle werden hier in Form von SysML-Aktivitätsdiagrammen (engl.: *Activity Diagram* (AD)) [OMG17a] spezifiziert. Die Testfälle werden so generiert, dass sie eine maximale C_2c-Pfadüberdeckung [Lig09] erfüllen. Eine hundertprozentige Pfadüberdeckung ist nicht immer möglich, da auch korrekte Diagramme invalide Pfade enthalten können. Bei einem invaliden Pfad schließen sich die Bedingungen an verschiedenen Entscheidungsknoten des Pfades aus, wodurch es keine Eingabe geben kann, die zu einem Durchlauf dieses Pfades führt. Die verwendeten Aktivitätsdiagramme folgen Modellierungsregeln, wodurch die folgenden Eigenschaften garantiert werden:

- Anforderungen in Textform lassen sich mit den Aktionen der Aktivitätsdiagramme verknüpfen.
- Eine Funktionsschnittstelle in Form von Eingabe- und Ausgabeinformationen lässt sich aus dem Diagramm ableiten.
- Diese Eingabe- und Ausgabeinformationen haben bei Betrachtung aller Diagramme einen global eindeutigen Namen.

Im Zuge des Projekts wurde ein projektspezifisches Artefaktmodell erstellt, welches in Listing 10.31 dargestellt ist. Mit dessen Hilfe konnten zwei wichtige Anforderungen an das Projekt erfüllt werden, welche im Nachgang erläutert werden. Die dargestellten Erweiterungen des Artefaktmodells wurden mit Fokus auf diese Anwendungsfälle vorgenommen, d.h., einige Konzepte wurden bewusst weggelassen oder aber in stark abstrahierter Form dargestellt.

CD
«AM4A»

```
class ActivityDiagram extends Artifact {}

class ADAction{}

association activities [1] ActivityDiagram -> ADAction [*];

class Information {
  String name;
}

association input [1] ActivityDiagram -> Information [*];

association output [1] ActivityDiagram -> Information [*];

class Requirement extends Artifact {
  String name;
  String text;
}

association linkedTo [*] Requirement -> ADAction [1];

class TestCase extends Artifact {}

association correspondsToPath
  [1] TestCase -> ADAction [*] <<ordered>>;
```

Listing 10.31: Projektspezifische Erweiterung des Artefaktmodells bei der Testfallgenerierung im Automotivekontext.

Aktivitätsdiagramme, Anforderungen und Testfälle werden in diesem Kontext als Artefakt angesehen. Durch Aktivitätsdiagramme wird der Ablauf von Funktionen in Form von durchzuführenden Aktionen beschrieben. Ein- und Ausgaben der Funktion werden durch das Informationskonzept modelliert. Anforderungen lassen sich mit den Aktionen der Diagramme verknüpfen und jeder Testfall entspricht einem Pfad durch das Aktivitätsdiagramm, der vereinfacht als Liste von Aktionen dargestellt wird. Tatsächlich werden bei dem verwendeten Generator Pfade als Abfolge von Kanten definiert [Mil17], sodass hier unter Umständen verschiedenen Testfälle mit einer identische Abfolge von Aktionen verlinkt sein können. Die Testfalleingaben (hier nicht modelliert) würden sich in solchen Fällen jedoch unterscheiden.

Der erste Anwendungsfall, der durch eine OCL-basierte Analyse beschrieben und durchgeführt werden kann, ist das Tracing (siehe Abschnitt 3.2) von Anforderungen zu Testfällen. Für die am Projekt beteiligten Testingenieure war entscheidend, dass die textuellen und mit den Modellen verknüpften Anforderungen adäquat verifiziert werden. Zu diesem Zweck sollten jedem Testfall die entsprechenden Anforderungen zugeordnet werden können. In Listing 10.32

wird zu diesem Zweck die abgeleitete Assoziation `trace` hinzugefügt, welche durch die dargestellte OCL-Berechnungsvorschrift spezifiziert wurde.

CD «AM4A»

```
1  association /trace [*] TestCase -> Requirement [*];
2
3  context TestCase inv:
4    trace == correspondsToPath.linkedTo;
```

Listing 10.32: Die OCL-basierte Analyse ermöglicht das Tracing von Anforderungen zu Testfällen.

Der zweite Anwendungsfall, der durch eine OCL-basierte Analyse beschrieben und durchgeführt werden kann, ist die Erstellung einer Funktionsarchitektur. Zu diesem Zweck müssen die Beziehungen zwischen den Funktionsmodellen in Form der Aktivitätsdiagramme erhoben werden. In Listing 10.33 wurden aus diesem Grund zwei abgeleitete Assoziationen hinzugefügt: `sendsTo` und `receivesFrom`.

CD «AM4A»

```
1  association /sendsTo
2    [*] ActivityDiagram -> ActivityDiagram [*];
3
4  association /receivesFrom
5    [*] ActivityDiagram -> ActivityDiagram [*];
6
7  context ActivityDiagram inv:
8    sendsTo == {a in ActivityDiagram |
9                a.input.removeAll(output) != a.input};
10
11 context ActivityDiagram inv:
12   receivesFrom ==
13     {a in ActivityDiagram | a.sendsTo.contains(this)};
```

Listing 10.33: Die OCL-basierte Analyse ermöglicht die Rekonstruktion der Funktionsarchitektur.

Während erstere alle ADs mit dem Quell-AD verknüpft, an die das Quell-AD Informationen versendet, beschreibt `receivesFrom` den umgekehrten Fall. Durch die OCL-Berechnungsvorschrift in Zeile 7 wird die Berechnung der `sendsTo`-Beziehung spezifiziert. Es werden alle ADs verknüpft, deren Eingaben mindestens eine Ausgabeinformation des Quell-ADs beinhalten. Zur Spezifikation der Rückrichtung wurde in der zweiten Berechnungsvorschrift die `sendsTo`-Beziehung genutzt.

Tatsächlich lässt sich die Modellschnittstelle nicht direkt aus den verwendeten Aktivitätsdiagrammen ablesen. Der Grund dafür ist, dass bestehende, projektspezifische Modellierungsrichtlinien nicht beliebig angepasst werden konnten. Die Schnittstelle, bestehend aus den Ein- und

Ausgaben der Funktion, lässt sich aber dennoch ableiten. Die Ableitung der Modellschnittstelle lässt sich ebenfalls mit Hilfe eines Artefaktmodells beschreiben, wird aber an dieser Stelle nicht weiter betrachtet. Des Weiteren können Aktionen des Aktivitätsdiagramms durch eigene Aktivitätsdiagramme dargestellt werden, wodurch hierarchisch verschachtelte Aktivitätsdiagramme entstehen. Dies lässt sich durch eine M2M-Transformation jedoch auflösen, weshalb hierarchische Beziehungen ebenfalls nicht Teil der in Listing 10.31 dargestellten Erweiterung des Artefaktmodells sind.

Kapitel 11.

Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Konzept zur artefaktbasierten Analyse von MDD-Projekten vorgestellt. In Abschnitt 11.1 werden abschließend die wesentlichen Beiträge der Arbeit zusammengefasst und in Abschnitt 11.2 wird auf offene, anknüpfende Forschungsfragestellungen eingegangen.

11.1. Zusammenfassung

MDD hilft dabei, die Komplexität bei der Softwareentwicklung zu beherrschen und die konzeptionelle Lücke zwischen Problem- und Lösungsdomäne zu schließen. Durch die Verwendung von Modellen und den Einsatz von MDD-Werkzeugen lässt sich der Softwareentwicklungsprozess zumindest teilweise automatisierten. Gerade bei Einsatz von MDD-Techniken in großen Entwicklungsprojekten kann es durch die große Menge verschiedener Artefakte, die Anzahl verschiedener Artefaktarten und die Verwendung von MDD-Werkzeugen zu unübersichtlichen Projektsituationen kommen. Dies kann beispielsweise zu einer schlechten Wartbarkeit oder einem ineffizienten Buildprozess führen.

Das Ziel dieser Arbeit war die Entwicklung von Konzepten, Methoden und Werkzeugen zur artefaktbasierten Analyse von modellgetriebenen Softwareentwicklungsprojekten. Der Begriff der artefaktbasierten Analyse wurde in dieser Arbeit zur Bezeichnung einer Reverse-Engineering-Technik genutzt, mit deren Hilfe wiederholbare und automatisierte Analysen von Artefaktstrukturen möglich sind. Bei der artefaktbasierten Analyse werden zunächst die im untersuchenden Projekt vorhandenen Artefakte und deren Beziehungen extrahiert, wodurch ein Verständnis des Projekts möglich wird und zuvor spezifizierte Projektanalysen durchgeführt werden können. Um ein einheitliches Verständnis des in dieser Arbeit betrachteten Umfangs von MDD-Projekten herbeizuführen, wurden in Kapitel 2 zunächst Grundlagen aus diesem Bereich vorgestellt. Anschließend wurden in Kapitel 3 Analysetechniken zur Analyse von Softwareentwicklungsprojekten betrachtet, die sich auch zur Analyse von MDD-Projekten eignen. Der Lösungsansatz, grundlegende Begriffsdefinitionen und Anforderungen an die zu unterstützenden Analysen, an die Modellierungstechnik und an die Werkzeugunterstützung wurden anschließend in Kapitel 4 behandelt.

Zur Definition von Artefakten und deren Beziehungen, Artefakt- und Beziehungsarten sowie zur Spezifikation von Analysen wurde in Kapitel 5 eine auf der UML/P basierende Modellierungstechnik definiert. Diese ermöglicht die Spezifikation von projektspezifischen Artefaktmodellen durch CD- und OCL-Anteile. Als Teil der Modellierungstechnik wurde unter Verwendung dieser Sprachen ein projektübergreifend wiederverwendbarer AM-Kern spezifiziert. Außerdem lassen sich Analysespezifikationen ebenfalls durch CDs und OCL-Ausdrücke angeben. Demgegenüber werden Artefaktdaten, welche die aktuelle Projektsituation widerspiegeln, durch ODs definiert. Die Wahl der Modellierungstechnik erlaubt eine Prüfung der Konsistenz zwischen AM und Artefaktdaten. Darüber hinaus werden die Modelle in einer für Menschen gut lesbaren Form spezifiziert, können aber auch maschinell durch Werkzeuge weiterverarbeitet werden.

Ein konkretes AM für MontiCore-basierte Entwicklungsprojekte wurde in Kapitel 6 vorgestellt. Dieses AM kann als eigener Beitrag angesehen werden, da dort Artefakt- und Beziehungsarten sowie Bedingungen an solche Projekte definiert wurden, was zu einem besseren Verständnis dieser Projekte beiträgt. Weiterhin wurde dieses AM für die Definition und Durchführung von Analysen in realen Projekten in Kapitel 10 verwendet. Außerdem konnten die Erfahrungen bei der Erstellung des AMs zur Definition einer Methodik für die Erstellung projektspezifischer AMe und für die Spezifikation von Analysen als Teil einer Gesamtmethodik genutzt werden. Der dritte Teil der in Kapitel 7 vorgestellten Gesamtmethodik beschreibt das Vorgehen zur Durchführung von artefaktbasierten Analysen selbst. Die artefaktbasierte Analyse besteht aus fünf Teilschritten, die, ausgehend von einem MDD-Projekt und einem zugehörigen AM, die Durchführung von spezifizierten Analysen ermöglichen. Um die Durchführung von artefaktbasierten Analysen zu unterstützen, wurde mit MontiArT eine Werkzeugkette entwickelt, deren Werkzeuge einzelne (Teil-)Schritte der artefaktbasierten Analyse automatisiert durchführen können. Neben diesen Werkzeugen wurden auch die entwickelten Werkzeuge zur manuellen Exploration und zur Visualisierung von Artefaktdaten in Kapitel 8 vorgestellt. Darüber hinaus wurde in Kapitel 9 die Anwendung und Anpassung von MontiArT beschrieben.

Durch die erbrachten Ergebnisse trägt diese Arbeit dazu bei, die steigende Komplexität großer MDD-Projekte beherrschbar zu machen, indem sich sowohl Artefakt- und Beziehungsarten explizit modellieren lassen als auch eine Möglichkeit zur manuellen und automatisierten Analyse von MDD-Projekten geschaffen wurde. Verborgene Beziehungen lassen sich offen legen und direkt überprüfen, wodurch sich erst die Möglichkeit für Korrekturen und Optimierungen im Projekt ergeben. Weiterhin wurde durch die Definition grundlegender Begriffe und einer Methodik zur Analyse von MDD-Projekten sowie der Entwicklung einer konfigurierbaren Werkzeugkette, deren Werkzeuge sich in mehreren Kontexten wiederverwenden lassen, der Grundstein für weitere Forschungsarbeiten in diesem Bereich gelegt.

11.2. Ausblick

Die vorgestellte Methodik bietet zusammen mit der Werkzeugkette MontiArT bereits umfangreiche Möglichkeiten zur artefaktbasierten Analyse modellgetriebener Softwareentwicklungsprojekte. Nichtsdestotrotz wurden einige interessante offene Fragestellungen identifiziert, mit deren Verfolgung die vorgestellte Arbeit weitergeführt werden kann. Diese werden im Folgenden vorgestellt.

Die in dieser Arbeit dargestellte Methodik zur artefaktbasierten Analyse modellgetriebener Softwareentwicklungsprojekte betrachtet Daten immer zu genau einem Zeitpunkt. Bei der agilen, modellbasierten Softwareentwicklung ist die Betrachtung von Daten zu verschiedene Zeitpunkten ebenfalls interessant, da so Aussagen über die Veränderung zwischen zwei Entwicklungszeitpunkten getroffen werden können. Kleine Änderungen am Generator können bereits zu großen Änderungen am generierten Produkt führen, die bei komplexen Prozessen nicht immer direkt ersichtlich sind. Ein erster Ansatz für den Vergleich von Daten mehrerer Zeitpunkte wurde in der Arbeit [Buc17] vorgestellt. Hier wurde ein Werkzeug zur Identifikation von Unterschieden in Objektdiagrammen entwickelt, welches sich auf die Artefaktdaten von MontiArT anwenden lässt. Offen ist hingegen die Frage, ob und wie sich die in dieser Arbeit vorgestellten Analysetechniken auf mehrere, kombinierte und durch die Zeit veränderte Datensätze anwenden lassen.

Weiterhin liegt der Fokus dieser Arbeit darauf, Beziehungen zwischen Artefakten für die Analyse in modellgetriebenen Softwareentwicklungsprojekten zu nutzen. Es ist durchaus denkbar, auch artefaktinterne Elemente bei Analysen stärker mit einzubeziehen, um ein vollständiges Bild des Gesamtprojekts zu erhalten. Im vorgestellten Artefaktmodell wurden innere Elemente nur in Ausnahmefällen und in geringer Detailstufe betrachtet. So wurden zwar Typen definiert, nicht aber zwischen speziellen Typen (*Interface*, *Klasse*, *Enum*) unterschieden. Darüber hinaus werden andere innere Elemente, wie beispielsweise Methoden, überhaupt nicht betrachtet. Eine interessante Fragestellung ist an dieser Stelle, ob sich mit einer Erweiterung der vorgestellten Methodik solche Informationen bei Analysen mitbetrachtet lassen, ohne dass die Komplexität der Analyseartefakte (Größe des Artefaktmodells, Menge der zugehörigen Daten, Anzahl der Analysen pro Projekt) zu stark ansteigt.

Neben der detaillierten Betrachtung von artefaktinternen Elementen gibt es ein weiteres Themengebiet, dessen Einbeziehung den Umfang der Analysedaten erhöht. So ist die Betrachtung von Zwischenergebnissen, welche nicht als Artefakt gespeichert werden, sondern nur flüchtig im modellgetriebenen Entwicklungsprozess vorkommen, ebenfalls von Interesse. Die Symboltabelle und der AST existieren nur bei Ablauf eines Werkzeugs im Speicher. Eine Betrachtung dieser Datenstrukturen zur Laufzeit würde ein feingranulareres Tracing ermöglichen, wodurch sich beispielsweise exakt bestimmen ließe, welche Elemente der Eingabemodelle zur Erstellung einer generierten Datei beigetragen haben. Zu beachten ist hier, dass sich die Datenstrukturen durch die Verwendung von M2M-Transformationen, die innerhalb des Werkzeugs durchgeführt werden, während der Ausführung des Werkzeugs ändern können.

Das Zusammenspiel von Werkzeugen und Artefakten ist bei der artefaktbasierten Analyse von besonderem Interessen. Allerdings wurde Werkzeugkopplung in dieser Arbeit nur im Zuge von automatisierten Buildprozessen in MDD-Projekten betrachtet. Es existieren aber auch andere Arten von Beziehungen zwischen Artefakten und Werkzeugen, die als Ergänzung zu dieser Arbeit bei artefaktbasierten Analysen mitbetrachtet werden könnten. Außerhalb eines automatisierten Buildprozesses verwalten Werkzeuge zugehörige Datenbestände, deren Einheiten als Artefakte angesehen werden können. Eine automatisierte Synchronisation zwischen den Artefakten verschiedener Werkzeuge sowie Werkzeuge die sich gegenseitig Aufrufen sind ebenfalls möglich. Weiterhin können sich mehrere Werkzeuge ein und denselben Datenbestand teilen. Diese verschiedenen Interaktionsarten [BJL+15] könnten ebenfalls durch geeignete Erweiterungen des vorgestellten Ansatzes in zukünftigen Arbeiten mitbetrachtet werden.

Der letzte, hier vorgestellte Punkt des Ausblicks betrifft den gewählten Modellierungsansatz. Die Erfahrungen aus der Modellierung des Artefaktmodells haben gezeigt, dass eine mehrstufige Modellierung mit den gewählten Modellierungssprachen umständlich sein kann. Gerade wenn große AMe erstellt und gewartet werden müssen, wird eine Optimierung der Modellierungstechnik interessant. Dazu könnten Ansätze aus dem Bereich Multi-Level-Modellierung [AK08] für die Modellierung von Artefaktzusammenhängen erprobt werden. Alternativ kann die Verwendung von speziell für die Artefaktmodellierung zugeschnittenen DSLs evaluiert werden, wozu eine entsprechende DSL entwickelt und angewendet werden müsste.

Literaturverzeichnis

[ABO+17] Nour Ali, Sean Baker, Ross O'Crowley, Sebastian Herold, and Jim Buckley. Architecture consistency: State of the practice, challenges and requirements. *Empirical Software Engineering*, pages 1–35, 2017.

[ABT10] Thorsten Arendt, Matthias Burhenne, and Gabriele Taentzer. Defining and Checking Model Smells: A Quality Assurance Task for Models based on the Eclipse Modeling Framework. In *9th BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)*, 2010.

[AHRW17] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54:2787 – 2805, 2010.

[AK03] C. Atkinson and T. Kuhne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20:36–41, 2003.

[AK08] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7:345–359, 2008.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-wesley, 2 edition, 2007.

[AMS17] Apache Maven Shade Plugin `https://maven.apache.org/plugins/maven-shade-plugin/`, visited: 20.05.2017.

[Atl17] Bamboo – Continuous Integration, Deployment und Release-Management `https://de.atlassian.com/software/bamboo`, visited: 01.12.2017.

[BA96] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[Bal11] Helmut Balzert. *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Springer, 2011.

[BCE+06] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A Manifesto for Model Merging. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*. ACM, 2006.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[Bec04] Kent Beck. *Extreme Programming Explained: Embrace Change*. addison-wesley, 2004.

[Béz05] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4:171–188, 2005.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273–280. IEEE, 2001.

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. Part of the Grand Challenges in Modeling (GRAND'17) Workshop, July 2017.

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.

[BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 53–70. Springer, July 2017.

[BHK+16] Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. View-based model-driven software development with ModelJoin. *Software & Systems Modeling*, 15, 2016.

[BHT+10] Steffen Becker, Michael Hauck, Mircea Trifu, Klaus Krogmann, and Jan Kofron. Reverse Engineering Component Models for Quality Predictions. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 194–197. IEEE, 2010.

[BJL+15] Barrett Bryant, Jean-Marc Jézéquel, Ralf Lämmel, Marjan Mernik, Martin Schindler, Friedrich Steinmann, Juha-Pekka Tolvanen, Antonio Vallecillo, and Markus Völter. *Globalized Domain Specific Language Engineering*, pages 43–69. Springer, 2015.

[BJRV05] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the Large and Modeling in the Small. In *Model Driven Architecture*, pages 33–46. Springer, 2005.

[BJV04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the need for megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[BM01] Oliver Bastert and Christian Matuszewski. *Layered Drawings of Digraphs*, pages 87–120. Springer, 2001.

[Bro13] Manfred Broy. A Logical Approach to Systems Engineering Artifacts and Traceability: From Requirements to Functional and Architectural Views. *Engineering Dependable Software Systems*, 34, 2013.

[Bro18] Manfred Broy. A logical approach to systems engineering artifacts: semantic relationships and dependencies beyond traceability—from requirements to functional and architectural views. *Software & Systems Modeling*, 17:365–393, 2018.

[BRW17] Tim Bolender, Bernhard Rumpe, and Andreas Wortmann. Investigating the Effects of Integrating Handcrafted Code in Model-Driven Engineering. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[BSB08] Christoph Bommer, Markus Spindler, and Volkert Barr. *Softwarewartung: Grundlagen, Management und Wartungstechniken*. dpunkt.verlag, 2008.

[BTR08] Steffen Becker, Mircea Trifu, and Ralf Reussner. Towards Supporting Evolution of Service-Oriented Architectures through Quality Impact Prediction. In *Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pages 77–81. IEEE, 2008.

[Buc17] Lennart Bucher. Ein konfigurierbares Tool zur Identifikation von strukturellen Unterschieden in Objektdiagrammen. Bachelorarbeit, RWTH Aachen University, 2017.

[CAvdSA11] Trosky B. Callo Arias, Pieter van der Spek, and Paris Avgeriou. A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, 16:544–586, 2011.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CC90] E. J. Chikofsky and J. H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 1990.

[CCF+15a] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CCF+15b] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe. Globalizing Domain-Specific Languages (Dagstuhl Seminar 14412). *Dagstuhl Reports*, 4(10):32–50, 2015.

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*, 2003.

[CHGZ12] Jane Cleland-Huang, Orlean Gotel, and Andrea Zisman. *Software amd Systems Traceability*. Springer, 2012.

[Chr75] Nicos Christofides. *Graph Theory: An Algorithmic Approach (Computer Science and Applied Mathematics)*. Academic Press, Inc., 1975.

[CNSM13] J. Czerwonka, N. Nagappan, W. Schulte, and B. Murphy. CODEMINE: Building a Software Development Data Analytics Platform at Microsoft. *IEEE Software*, 30:64–71, 2013.

[Con16] Maximilian Conze. Dynamische, modellbasierte Abhängigkeitsanalyse von Java Systemen. Bachelorarbeit, RWTH Aachen University, 2016.

[CP13] G. Ann Campbell and Patroklos P. Papapetrou. *SonarQube in Action*. Manning Publications Co., 2013.

[CS14] Scott Chacon and Ben Straub. *Pro git*. Apress, 2014.

[Csi96] Mihaly Csikszentmihalyi. *Creativity: Flow and the Psychology of Discovery and Invention*. HarperCollins, 1996.

[Cuj17] Thomas Cujé. Verwendung von OCL für die Artefaktmodellierung. Bachelorarbeit, RWTH Aachen University, 2017.

[Cun92] Ward Cunningham. The WyCash Portfolio Management System. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, pages 29–30. ACM, 1992.

[Ded16] Jonas Dederichs. Web-based Visualisation of Artifact Structures in Model-driven Software Development. Bachelor's thesis, RWTH Aachen University, 2016.

[DHJ05] Jeremy Dick, Elizabeth Hull, and Ken Jackson. *Requirements Engineering*. Springer, 2 edition, 2005.

[Die06] Reinhard Diestel. *Graphentheorie*. Springer, 2 edition, 2006.

[Die12] Jens Dietrich. Upload Your Program, Share Your Model. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 21–22, 2012.

[DKM06] T. Dwyer, Y. Koren, and K. Marriott. IPSep-CoLa: An Incremental Procedure for Separation Constraint Layout of Graphs. *IEEE Transactions on Visualization and Computer Graphics*, 2006.

[dLG10] Juan de Lara and Esther Guerra. Deep Meta-modelling with MetaDepth. In *Objects, Models, Components, Patterns*, pages 1–20. Springer, 2010.

[DP09] Stephane Ducasse and Damien Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35:573–591, 2009.

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309. ACM, 2010.

[Ecl17] Eclipse `http://www.eclipse.org`, visited: 11.12.2017.

[EF17] C. Ebert and J. Favaro. Automotive Software. *IEEE Software*, 34:33–39, 2017.

[Fav06] Jean-Marie Favre. Megamodelling and Etymology. In *Transformation Techniques in Software Engineering*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[FB99] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FBB08] Ralph Foorthuis, Sjaak Brinkkemper, and Rik Bos. An Artifact Model for Projects Conforming to Enterprise Architecture. In *The Practice of Enterprise Modeling*, pages 30–46. Springer, 2008.

[FJE17] Fat Jar Eclipse Plug-In `http://fjep.sourceforge.net/`, visited: 30.12.2017.

[FKH17] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology*, 87:259–277, 2017.

[FKSJ14] H. Femmer, M. Kuhrmann, J. Stimmer, and J. Junge. Experiences from the Design of an Artifact Model for Distributed Agile Project Management. In *2014 IEEE 9th International Conference on Global Software Engineering*, pages 1–5, 2014.

[FN05] Jean-Marie Favre and Tam NGuyen. Towards a Megamodel to Model Software Evolution Through Transformations. *Electronic Notes in Theoretical Computer Science*, 127:59–74, 2005.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[Fow06] Martin Fowler. Continuous Integration `https://www.martinfowler.com/articles/continuousIntegration.html`, 2006.

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley, 2010.

[FPKB10] Daniel Méndez Fernández, Birgit Penzenstadler, Marco Kuhrmann, and Manfred Broy. A meta model for artefact-orientation: fundamentals and lessons learned in requirements engineering. In *International Conference on Model Driven Engineering Languages and Systems*, pages 183–197. Springer, 2010.

[FR07] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)*, pages 37–54, May 2007.

[Fre17] FreeMarker Java Template Engine `http://freemarker.org/`, visited: 04.12.2017.

[Gem17] The GEMOC Initiative `http://gemoc.org/`, visited: 23.10.2017.

[GG118] google-gson: A Java serialization/deserialization library to convert Java Objects into JSON and back `https://github.com/google/gson`, visited: 13.01.2018.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.

[GHK+15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, volume 580 of *Communications in Computer and Information Science*, pages 112–132. Springer, 2015.

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects*. Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.

[GIM13] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A Comparative Analysis of Software Architecture Recovery Techniques. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 486–496. IEEE Press, 2013.

[GJ14] Martin Gronemann and Michael Jünger. *Visualisierung verschachtelter Graphen als Landkarten*, pages 317–328. Springer, 2014.

[GJS+15] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Oracle America, Java SE 8 edition, 2015.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GLR17] Timo Greifenberg, Markus Look, and Bernhard Rumpe. Visualizing MDD Projects. In *Software Engineering Conference (SE'17)*, LNI P267, pages 101–104. Bonner Köllen Verlag, February 2017.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GMR15] Timo Greifenberg, Klaus Müller, and Bernhard Rumpe. Architectural Consistency Checking in Plugin-Based Software Systems. In *European Conference on Software Architecture Workshops (ECSAW'15)*, pages 58:1–58:7. ACM, 2015.

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, volume 254 of *LNI*, pages 141–156. Bonner Köllen Verlag, March 2016.

[Gra17] Graphviz `https://www.graphviz.org/`, visited: 12.11.2017.

[GS17] Victor Guana and Eleni Stroulia. End-to-end model-transformation comprehension through fine-grained traceability information. *Software & Systems Modeling*, pages 1–40, 2017.

[GTB18] Raffaela Groner, Matthias Tichy, and Steffen Becker. Towards Performance Engineering of Model Transformation. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 33–36. ACM, 2018.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[Hal77] Maurice Howard Halstead. *Elements of Software Science*. Elsevier New York, 1977.

[Hel17a] hello2morrow Sonargraph `https://www.hello2morrow.com/products/sonargraph`, visited: 02.11.2017.

[Hel17b] hello2morrow Sonargraph Explorer `https://www.hello2morrow.com/products/sonargraph/explorer`, visited: 02.11.2017.

[Her14] Christoph Herrmann. *Integrierte Software Engineering Services zur effizienten Unterstützung von Entwicklungsprozessen*. Aachener Informatik-Berichte, Software Engineering, Band 16. Shaker Verlag, 2014.

[HHRW15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, volume 1463 of *CEUR Workshop Proceedings*, pages 18–23, 2015.

[Hil14] Steffen Hillemacher. A Framework for Analysis and Visualization of Artifact Dependencies in Model Driven Development. Bachelor's thesis, RWTH Aachen University, 2014.

[Hil16] Steffen Hillemacher. Automated architecture checking for MontiCore-based software development projects. Master's thesis, RWTH Aachen University, 2016.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HLMSN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, volume 580 of *Communications in Computer and Information Science*, pages 45–66. Springer, 2015.

[HLMSN+15b] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015.

[HMSNR+15a] Robert Heim, Pedram Mir Seyed Nazari, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Modeling Robot and World Interfaces for Reusable Tasks. In *Intelligent Robots and Systems Conference (IROS'15)*, pages 1793–1798. IEEE, 2015.

[HMSNR15b] Katrin Hölldobler, Pedram Mir Seyed Nazari, and Bernhard Rumpe. Adaptable Symbol Table Management by Meta Modeling and Generation of Symbol Table Infrastructures. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 23–30. ACM, 2015.

[HMSNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.

[Hof13] Dirk W Hoffmann. *Software-Qualität*. Springer, 2013.

[Höl18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HSG11] Regina Hebig, Andreas Seibel, and Holger Giese. On the Unification of Megamodels. *Electronic Communications of the EASST*, 42, 2011.

[HUS17] HUSACCT System Documentation: Analyse Java & C-Sharp `https://github.com/HUSACCT/HUSACCT/blob/master/doc/system/analyse/HUSACCT%20Analyse%20System%20Documentation.pdf`, 02.11.2017.

[HZ12] Thomas Haitzer and Uwe Zdun. DSL-based Support for Semi-automated Architectural Component Model Abstraction Throughout the Software Lifecycle. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*, QoSA '12. ACM, 2012.

[IBM17] IBM Knowledge Center - Abhängigketisanalyse `https://www.ibm.com/support/knowledgecenter/de/SSWLGF_8.5.6/com.ibm.sr.doc/twsr_mansrvce_governanceuserguide05.html`, visited: 11.12.2017.

[IEE98] IEEE Standard for a Software Quality Metrics Methodology, 1998.

[IEE05] Software Maintenance. *IEEE Software*, 22:103–103, 2005.

[Int17a] IntelliJ IDEA `https://www.jetbrains.com/idea/`, visited: 11.12.2017.

[Int17b] Ecma International. Standard ECMA-404: The JSON Data Interchange Syntax, 2017. https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[Jac11] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT press, 2011.

[JFr17] Artifactory - Universal Artifact Repository Manager `https://www.jfrog.com/artifactory/`, visited: 14.09.2017.

[JUn18] JUnit website `http://junit.org/`, visited: 13.01.2018.

[Käs16] Johannes Dominik Kästle. Erweiterung der Artefaktstruktur-Analyse und Integration in die bestehende Infrastruktur. Bachelorarbeit, RWTH Aachen University, 2016.

[KB14] Marco Kuhrmann and Sarah Beecham. Artifact-based software process improvement and management: a method proposal. In *Proceedings of the 2014 International Conference on Software and System Process*, pages 119–123. ACM, 2014.

[KFG13] M. Kuhrmann, D. M. Fernández, and M. Gröber. Towards Artifact Models as Process Interfaces in Distributed Software Projects. In *2013 IEEE 8th International Conference on Global Software Engineering*, pages 11–20, 2013.

[KH10] Maximilian Koegel and Jonas Helming. EMFStore: A Model Repository for EMF Models. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10. ACM, 2010.

[KMS+17] Stefan Kriebel, Vincent Moyses, Georg Strobl, Johannes Richenhagen, Phillip Orth, Stefan Pischinger, Christoph Schulze, Timo Greifenberg, and Bernhard Rumpe. The Next Generation of BMW's Electrified Powertrains: Providing Software Features Quickly by Model-Based System Design. In *26th Aachen Colloquium Automobile and Engine Technology*, October 2017.

[KMS+18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[Küh06] Thomas Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5:369–385, 2006.

[KV10] Lennart C.L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463. ACM, 2010.

[KWK10] Johan Kraft, Anders Wall, and Holger Kienle. Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects. In *Proceedings of the First International Conference on Runtime Verification (RV 2010)*. Springer, November 2010.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, 2004.

[Lee08] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008.

[Leh15] Steffen Lehnert. *Multiperspective Change Impact Analysis to Support Software Maintenance and Reengineering*. PhD thesis, University of Hamburg, 2015.

[Lig09] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Springer, 2009.

[Lil16] Carola Lilienthal. *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*. dpunkt, 2016.

[LL13] Jochen Ludewig and Horst Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2013.

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE*. Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.

[LYBB15] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*. Oracle America, Java SE 8 edition, 2015.

[Mai15] Ben Mainz. Modellbasierte Abhängigkeitsanalyse von Generatorartefakten. Bachelorarbeit, Fachhochschule Aachen, 2015.

[Mas17] Massey Architecture Explorer 2.2.0 `http://xplrarc.massey.ac.nz/`, visited: 02.11.2017.

[MB17] Sander Mak and Paul Bakker. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications*. O'Reilly Media, 2017.

[McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on software Engineering*, pages 308–320, 1976.

[MCG18] MontiCore - Getting Started `http://www.monticore.de/gettingstarted/`, visited: 06.10.2018.

[Mel12] Volodymyr Melymuka. *TeamCity 7 Continuous Integration Essentials*. Packt Publishing Ltd, 2012.

[MFP15] Daniel Méndez Fernández and Birgit Penzenstadler. Artefact-based requirements engineering: the AMDiRE approach. *Requirements Engineering*, 20:405–434, 2015.

[MHDZ16] Markus Müller, Klaus Hörmann, Lars Dittmann, and Jörg Zimmer. *Automotive SPICE in der Praxis: Interpretationshilfe für Anwender und Assessoren*. dpunkt.verlag, 2 edition, 2016.

[MID17] MID - Analysemöglichkeiten auf Basis von Traceability Informationen `http://blog.mid.de/analysemoeglichkeiten-auf-basis-von-traceability-informationen`, visited: 11.12.2017.

[Mil17] Ivana Miljkovic. Generating Test Cases from Activity Diagrams in an industrial Context. Master's thesis, RWTH Aachen University, 2017.

[MNS01] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, 27:364–380, 2001.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MSN17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[MSNRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, volume 254 of *LNI*, pages 133–140. Bonner Köllen Verlag, March 2016.

[Mur11] Katherine Murray. *Microsoft Office 365: Connect and Collaborate Virtually Anywhere, Anytime*. Microsoft Press, 2011.

[MW16] Manfred Moser and Dulani Wallace. Repository Management with Nexus, 2016.

[NLG+15] Ana Nicolaescu, Horst Lichter, Artjom Göringer, Peter Alexander, and Dung Le. The ARAMIS Workbench for Monitoring, Analysis and Visualization of Architectures based on Run-time Interactions. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, page 57. ACM, 2015.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, volume 1118 of *CEUR Workshop Proceedings*, pages 15–24, 2013.

[OMG14a] Object Management Group. Model Driven Architecture (MDA) MDA Guide, 2014.

[OMG14b] Object Management Group. Object Constraint Language (OCL, 2014. http://www.omg.org/spec/OCL/2.4.

[OMG15] Object Management Group. Unified Modeling Language (UML), 2015. http://www.omg.org/spec/UML/.

[OMG16] Object Management Group. OMG Meta Object Facility (MOF) Core Specification, November 2016. http://www.omg.org/spec/MOF/2.5.1/PDF/.

[OMG17a] Object Management Group. OMG Systems Modeling Language (OMG SysML), 2017. http://www.omg.org/spec/SysML/1.5/.

[OMG17b] Object Management Group. OMG `http://www.omg.org/`, visited: 02.11.2017.

[Ora16] The Java Tutorials - What is a Package? `http://docs.oracle.com/javase/tutorial/java/concepts/package.html`, visited: 19.05.2016.

[Ora18] FileVisitor (Java Platform SE 8) `https://docs.oracle.com/javase/8/docs/api/index.html?java/nio/file/FileVisitor.html`, visited: 02.01.2018.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[PCSF11] C. Michael Pilato, Ben Collins-Sussman, and Brian Fitzpatrick. *Version Control with Subversion*. Ö'Reilly Media, Inc.", 2011.

[PKB13] L. Pruijt, C. Köppe, and S. Brinkkemper. Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparative Study of Tool Support. In *2013 IEEE International Conference on Software Maintenance*, 2013.

[PKvdWB14] Leo J. Pruijt, Christian Köppe, Jan Martijn van der Werf, and Sjaak Brinkkemper. HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 851–854. ACM, 2014.

[PR09] Gerold Patzak and Günter Rattay. *Projektmanagement: Leitfaden zum Management von Projekten, Projektportfolios, Programmen und projektorientierten Unternehmen*. Linde Verlag, 2009.

[PvDB12] M. C. Platenius, M. von Detten, and S. Becker. Archimetrix: Improved Software Architecture Recovery in the Presence of Design Deficiencies. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 255–264, 2012.

[PvdW15] Leo Pruijt and Jan Martijn E. M. van der Werf. Dependency Types and Subtypes in the Context of Architecture Reconstruction and Compliance Checking. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15. ACM, 2015.

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen*. Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.

[RF16] Chet Ramey and Brian Fox. *Bash Reference Manual*. Free Software Foundation, 4.4 edition, 2016.

[Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex*. Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRS+16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen, and Michael von Wenckstern. Test-driven Semantical Similarity Analysis for Software Product Line Extraction. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 174–183. ACM, September 2016.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArc-Automaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[SB02] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2002.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[Sei03] Edwin Seidewitz. What Models Mean. *IEEE software*, 20:26–32, 2003.

[Sei13] Andreas Seibel. *Traceability and Model Management with Executable and Dynamic Hierarchical Megamodels*. doctoralthesis, Universität Potsdam, 2013.

[Sha14] Kishori Sharan. *Beginning Java 8 Fundamentals: Language Syntax, Arrays, Data Types, Objects, and Regular Expressions*. Apress, 2014.

[SI16] M. Shahid and S. Ibrahim. Change Impact Analysis with a Software Traceability Approach to Support Software Maintenance. In *2016 13th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pages 391–396, 2016.

[SJ16] William E. Shotts Jr. *The Linux Command Line*. No Starch Press, Third Internet Edition edition, 2016.

[SJ17] William E. Shotts Jr. Writing Shell Scripts - Lesson 14 `http://linuxcommand.org/lc3_wss0140.php`, visited: 27.05.2017.

[SKCM16] Rick Salay, Sahar Kokaly, Marsha Chechik, and Tom Maibaum. Heterogeneous Megamodel Slicing for Model Evolution. In *ME@ MODELS*, pages 50–59, 2016.

[Sma11] John Ferguson Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. O'Reilly, 2011.

[SME09] Rick Salay, John Mylopoulos, and Steve Easterbrook. Using Macromodels to Manage Collections of Related Models. In *Advanced Information Systems Engineering*, pages 141–155. Springer, 2009.

[SMS16] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make: A Program for Directing Recompilation*. Free Software Foundation, May 2016.

[Sof17] Software Modeling and Development `https://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/`, visited: 04.11.2017.

[Sou16] Nikolaos Souris. Modeling of Software Tool Chains. Bachelor's thesis, RWTH Aachen University, 2016.

[SPBS15] Jocelyn Simmonds, Daniel Perovich, María Cecilia Bastarrica, and Luis Silvestre. A megamodel for Software Process Line modeling and evolution. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 406–415. IEEE, 2015.

[SSC96] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring Compliance of a Software System with Its High-level Design Models. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 387–396. IEEE Computer Society, 1996.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. {Springer-Verlag, Wien}, 1973.

[Str17a] Structure101 `https://structure101.com/`, visited: 02.11.2017.

[Str17b] Structure101 - Dependency Matrix `http://structure101.com/legacy/slice-and-tangle/`, visited: 02.11.2017.

[TELW14] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software & Systems Modeling*, 13, 2014.

[TFB+17] E. H. B. Toure, I. Fall, A. Bah, M. S. Camara, and M. Ba. Consistency Preserving for Evolving Megamodels through Axiomatic Semantics. In *2017 Intelligent Systems and Computer Vision (ISCV)*, pages 1–8, 2017.

[The17] Oliver Heinrich Thelen. Konformitätsprüfung von Modulbeziehungen im Kontext der Artefaktanalyse. Bachelorarbeit, RWTH Aachen University, 2017.

[TMD09] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[TNL17] Jan Thomas, Ana Nicolaescu, and Horst Lichter. Static and Dynamic Architecture Conformance Checking: A Systematic, Case Study-Based Analysis on Tradeoffs and Synergies. In *5th International Workshop on Quantitative Approaches to Software Quality*, 2017.

[TOG18] The Open Group Base Specifications Issue 7 `http://pubs.opengroup.org/onlinepubs/9699919799/`, visited: 11.01.2018.

[Tra17] Travis CI - Test and Deploy with Confidence `https://travis-ci.com/`, visited: 01.12.2017.

[vDB11] Markus von Detten and Steffen Becker. Combining Clustering and Pattern Detection for the Reengineering of Component-based Software Systems. In *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, QoSA-ISARCS '11, pages 23–32. ACM, 2011.

[vHRH+09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Technical report, Kiel University, 2009.

[vHWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, 2012.

[Vis17] Visual Studio-IDE `https://www.visualstudio.com/`, visited: 11.12.2017.

[Voe13] Markus Voelter. *Language and IDE Modularization and Composition with MPS*, pages 383–430. Springer, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[vSBCR02] Rini van Solingen, Vic Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal Question Metric (GQM) Approach. *Encyclopedia of Software Engineering*, 2002.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[WL07] R. Wettel and M. Lanza. Visualizing Software Systems as Cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99, 2007.

[WLR11] Richard Wettel, Michele Lanza, and Romain Robbes. Software Systems As Cities: A Controlled Experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 551–560. ACM, 2011.

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling*. Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.

[Yak17a] YAKINDU Traceability `https://www.itemis.com/en/yakindu/traceability/`, visited: 02.11.2017.

[Yak17b] YAKINDU Traceability - Visualize the relation between artifacts with YAKINDU Traceability `https://blogs.itemis.com/en/visualise-`

`the-relation-between-artifacts-with-yakindu-traceability`, visited: 02.11.2017.

Anhang A.

Markierungen in Abbildungen und Listings

In diesem Kapitel wird eine Übersicht über die Markierungen und Stereotypen, die in dieser Arbeit in Abbildungen und Listings zur Angabe der verwendeten (Modellierungs-) Sprache bzw. des Verwendungszwecks genutzt werden, dargestellt.

Markierung	Beschreibung
AD	Activity Diagram
CD	Class Diagram
Conf	`ODMerger` Configuration
FTL	FreeMarker-Template
Java	Java-Quellcode
JSON	JSON-Quellcode
Make	Makefile
OCL	Object Constraint Language
OD	Object Diagram
SH	Shell.Script
TD	Tagdefinition
TS	Tagschema

Tabelle A.1.: Die in Abbildungen und Listings verwendeten Markierungen. Markierungen geben die verwendete Sprache an.

Markierung	Stereotype	Beschreibung
CD	«AM»	Artefaktmodell
	«AM4A»	Ergänzung des Artefaktmodells zu Analysezwecken
OD	«ArtData»	Artefaktdaten
	«DiffArc»	Differenzarchitektur
	«IstArc»	Ist-Architektur
	«MD»	Moduldefinition
	«Report»	Report im OD-Format
	«SollArc»	Soll-Architektur

Tabelle A.2.: Die in Abbildungen und Listings verwendeten Stereotypen. Stereotypen sind Markierungen zugeordnet. Sie geben den Verwendungszweck an.

Anhang B.

Tracing der Anforderungen

Die in Abschnitt 4.5 definierten Anforderungen werden in verschiedenen Abschnitten der Arbeit referenziert. Die folgenden Tabellen erlauben ein Tracing der Anforderungen zu den entsprechenden Abschnitten. Im Fall von Unteranforderungen (siehe beispielsweise Anforderung *MRQ-3*) wird die übergeordnete Anforderung nicht in der entsprechenden Tabelle aufgeführt.

Anforderung	**Erfüllung beschrieben in**
ARQ-1	Abschnitt 8.2
ARQ-2	Abschnitt 7.2.2
ARQ-3	Abschnitt 7.2.2
ARQ-4	Abschnitt 7.2.3
ARQ-5	Abschnitt 7.2.3
ARQ-6	Abschnitt 7.2.4

Tabelle B.1.: Zuordnung von Anforderungen an Analyseverfahren zu den Abschnitten, welche die Anforderungserfüllung beschreiben.

Anforderung	Erfüllung beschrieben in
MRQ-1	Abschnitt 5.1, Abschnitt 7.1
MRQ-2	Abschnitt 5.1.2, Abschnitt 7.2.5, Abschnitt 7.6
MRQ-3.1	Abschnitt 5.2
MRQ-3.2	Abschnitt 7.1
MRQ-4	Abschnitt 7.1
MRQ-5	Abschnitt 5.1.2
MRQ-6	Abschnitt 5.1.2
MRQ-7	Abschnitt 7.2
MRQ-8	Abschnitt 7.2.3, Abschnitt 7.2.4, Abschnitt 7.2.5
MRQ-9	Abschnitt 7.5

Tabelle B.2.: Zuordnung von Anforderungen an die Modellierungstechnik zu den Abschnitten, welche die Anforderungserfüllung beschreiben.

Anforderung	Erfüllung beschrieben in
TRQ-1	Abschnitt 7.2, Abschnitt 8.1, Abschnitt 8.4.3, Abschnitt 8.5.1
TRQ-2	Einleitung Kapitel 8, Abschnitt 8.5.3
TRQ-3.1	Abschnitt 8.1, Abschnitt 8.1.1, Abschnitt 8.1.2
TRQ-3.2	Abschnitt 8.1.1
TRQ-3.3	Abschnitt 8.1.3
TRQ-4.1	Abschnitt 8.5.1, Abschnitt 8.5.5
TRQ-4.2	Abschnitt 7.2, Abschnitt 9.5, Abschnitt 9.5.2
TRQ-4.3	Abschnitt 9.5 Abschnitt 9.5.2

Tabelle B.3.: Zuordnung von Anforderungen an die Werkzeugunterstützung zu den Abschnitten, welche die Anforderungserfüllung beschreiben.

Anhang C.

Artefaktmodell für MontiCore-basierte MDD-Projekte

Das in diesem Kapitel dargestellte AM basieren stark auf[GHR17] und wurde lediglich an einigen Stellen leicht überarbeitet. Das AM aus [GHR17] umfasst zusätzlich Anteile zur Beschreibung von Maven-basierten Buildprozessen.

CD
«AM»

```
classdiagram ArtifactModel {

  // ____________________________________________________________
  // Artifact
  // ____________________________________________________________

  abstract class Artifact {
    String simpleName;
    String nameExtension;
    Date modified;
    /String name;
    /String fullName;
    /boolean isRoot;
  }

  context Artifact inv:
    name == ((nameExtension == "") ? simpleName :
      simpleName + "." + nameExtension);

  association /refersTo [*] Artifact -> Artifact [*];

  association /produces [*] Artifact -> Artifact [*];

  context Artifact a inv:
    !a.produces**.contains(a);
```

```
26
27  //_____________________________________________________________
28  // ArtifactContainer
29  //_____________________________________________________________
30
31  abstract class ArtifactContainer extends Artifact {}
32
33  composition contains
34    [0..1] ArtifactContainer (parent) -> Artifact [*];
35
36  context Artifact inv:
37    (isRoot <=> parent.isAbsent) &&
38    (!isRoot implies {p in parent** | p.isRoot}.size == 1);
39
40  context Artifact inv:
41    fullName == (isRoot ? "/" :
42                 (parent.isRoot ? "/" + name :
43                  parent.fullName + "/" + name));
44
45  context Artifact a, Artifact b inv:
46    (a.name == b.name && a.parent == b.parent)
47      implies a == b;
48
49  //_____________________________________________________________
50  // Directory
51  //_____________________________________________________________
52
53  class Directory extends ArtifactContainer {}
54
55  context Directory inv:
56    nameExtension == "";
57
58  context Artifact inv:
59    isRoot implies (this in Directory);
60
61  context Artifact inv:
62    isRoot <=> (simpleName == "/");
63
64  context Artifact inv:
65    name.contains("/") implies isRoot;
66
67  //_____________________________________________________________
68  // Archive
```

```
69  //_____________________________________________________________
70
71  class Archive extends ArtifactContainer {}
72
73  // ____________________________________________________________
74  // JavaArtifacts
75  // ____________________________________________________________
76
77  abstract class JavaArtifact extends Artifact {}
78
79  association reliesOnJavaArtifact
80    [*] JavaArtifact -> JavaArtifact [*];
81
82  context JavaArtifact inv:
83    refersTo.containsAll(reliesOnJavaArtifact);
84
85  // ____________________________________________________________
86  // JavaSourceFile
87  // ____________________________________________________________
88
89  class JavaSourceFile extends JavaArtifact {}
90
91  association imports [*] JavaSourceFile -> JavaArtifact [*];
92
93  association compiledTo
94    [1] JavaSourceFile -> JavaClassFile [*];
95
96  context JavaSourceFile inv:
97    reliesOnJavaArtifact.containsAll(imports);
98
99  context JavaSourceFile inv:
100   produces.containsAll(compiledTo);
101
102 context JavaSourceFile inv:
103   nameExtension == "java";
104
105 // ____________________________________________________________
106 // JavaClassFile
107 // ____________________________________________________________
108
109 class JavaClassFile extends JavaArtifact {}
110
111 context JavaClassFile inv:
```

```
112     nameExtension == "class";
113
114   // ____________________________________________________________
115   // JavaArchive
116   // ____________________________________________________________
117
118   class JavaArchive extends Archive {}
119
120   context JavaArchive inv:
121     nameExtension == "jar";
122
123   // ____________________________________________________________
124   // Package
125   // ____________________________________________________________
126
127   class Package {
128     String name;
129     String fullName;
130     /boolean isRoot;
131   }
132
133   association belongsTo [*] JavaArtifact -> Package [1];
134
135   composition subPackage
136     [0..1] Package (parent) -> (child) Package [*];
137
138   context Package inv:
139     !name.contains(".");
140
141   context Package inv:
142     (isRoot <=> parent.isAbsent) &&
143     (!isRoot implies {p in parent** | p.isRoot}.size == 1);
144
145   context Package inv:
146     !isRoot implies
147       fullName == parent.fullName + "." + name;
148
149   context Package a, Package b inv:
150     (a.parent == b.parent && a.name == b.name) implies a == b;
151
152   // ____________________________________________________________
153   // Type
154   // ____________________________________________________________
```

```
155
156  class Type {
157    String simpleName;
158    /String name;
159    /String fullName;
160    /boolean isInnerType;
161  }
162
163  association defines [*] JavaArtifact -> Type [*];
164
165  composition contains [1] Package -> Type [*];
166
167  composition
168    [0..1] Type (containingType) -> (innerType) Type [*];
169
170  association reliesOn [*] Type -> Type [*];
171
172  context Type inv:
173    (isInnerType <=> !containingType.isAbsent) &&
174    (isInnerType implies
175      {t in containingType** | t.isInnerType}.size == 1);
176
177  context Type inv:
178    name == (isInnerType ?
179      containingType.name + "." + simpleName : simpleName);
180
181  context Type inv:
182    fullName == (package.isRoot ? name :
183      package.fullName + "." + simpleName);
184
185  // ____________________________________________________________
186  // Relation between Java Artifacts and Java Types
187  // ____________________________________________________________
188
189  context JavaSourceFile inv:
190    defines == compiledTo.defines;
191
192  context JavaSourceFile inv:
193    {t in defines | !t.isInnerType}.size == 1;
194
195  context JavaArtifact a1, a2,
196    Type t1 in a1.defines, Type t2 in a2.defines inv:
197      t1.reliesOn.contains(t2) implies
```

```
198         (a1 == a2 || a1.reliesOn.contains(a2));
199
200   context JavaArtifact a, Type t in a.defines inv:
201     !t.isInnerType implies t.simpleName == a.simpleName;
202
203   context JavaArtifact inv:
204     parent.fullName.replaceAll("/", ".")
205       .endsWith(belongsTo.fullName);
206
207   context JavaSourceFile inv:
208     forall n in {t.fullName.replaceAll(".", "/") |
209       t in defines, !t.isInnerType}:
210         fullName.endsWith(n + ".java");
211
212   context JavaClassFile inv:
213     parent.fullName.replaceAll("/", ".")
214       .endsWith(defines.package.fullName);
215
216   context JavaClassFile inv:
217     {simpleName} == defines.name.replaceAll(".", "$");
218
219   //_____________________________________________________________
220   // Language
221   //_____________________________________________________________
222
223   class Language {
224     String name;
225   }
226
227   class Grammar {}
228
229   class CoCo {
230     String name;
231   }
232
233   association grammar [*] Language -> Grammar [1];
234
235   association coco [*] Language -> CoCo [*];
236
237   context Language inv:
238     name != "";
239
240   //_____________________________________________________________
```

```
241  // GrammarFile
242  //_____________________________________________________________
243
244  class GrammarFile extends Artifact {}
245
246  association defines [1] GrammarFile -> Grammar [1];
247
248  association includes [*] GrammarFile -> GrammarFile [*];
249
250  context GrammarFile inv:
251    nameExtension == "mc4";
252
253  context GrammarFile inv:
254    refersTo.containsAll(includes);
255
256  //_____________________________________________________________
257  // CoCo
258  //_____________________________________________________________
259
260  association implementedBy [*] CoCo -> JavaSourceFile [1..*];
261
262  //_____________________________________________________________
263  // ModelFile
264  //_____________________________________________________________
265
266  abstract class ModelFile extends Artifact {}
267
268  association conformsTo [*] ModelFile -> Language [*];
269
270  // _____________________________________________________________
271  // CDModelFile
272  // _____________________________________________________________
273
274  class CDModelFile extends ModelFile {}
275
276  association imports [*] CDModelFile -> Artifact [*];
277
278  association reliesOn [*] CDModelFile -> Artifact [*];
279
280  association defines [0..1] CDModelFile -> Type [*];
281
282  context CDModelFile inv:
283    nameExtension == "cd";
```

```
284
285   context CDModelFile inv:
286     reliesOn.containsAll(imports);
287
288   context CDModelFile inv:
289     refersTo.containsAll(reliesOn);
290
291   context CDModelFile m1, m2,
292     Type t1 in m1.defines, Type t2 in m2.defines inv:
293       t1.reliesOn.contains(t2) implies
294         (m1 == m2 || m1.reliesOn.contains(m2));
295
296   context CDModelFile m, JavaArtifact a,
297     Type t1 in m.defines, Type t2 in a.defines inv:
298       t1.reliesOn.contains(t2) implies m.reliesOn.contains(a);
299
300   // ____________________________________________________________
301   // System
302   // ____________________________________________________________
303
304   abstract class System {
305     String name;
306     String version;
307   }
308
309   association consistsOf [*] System -> Artifact [*];
310
311   // ____________________________________________________________
312   // Product
313   // ____________________________________________________________
314
315   class Product extends System {}
316
317   // ____________________________________________________________
318   // Tool
319   // ____________________________________________________________
320
321   class Tool extends System {}
322
323   // ____________________________________________________________
324   // Module
325   // ____________________________________________________________
326
```

```
327   class Module {
328     String name;
329   }
330
331   association module [*] System -> Module [*];
332
333   association subModule [*] Module -> Module [*];
334
335   association artifact [*] Module -> Artifact [*];
336
337   // ____________________________________________________________
338   // FreeMarkerTemplate
339   // ____________________________________________________________
340
341   class FreeMarkerTemplate extends Artifact {}
342
343   association reliesOnTemplate
344     [*] FreeMarkerTemplate -> FreeMarkerTemplate [*];
345
346   association reliesOnJavaArtifact
347     [*] FreeMarkerTemplate -> JavaArtifact [*];
348
349   association reliesOnTemplate
350     [*] JavaArtifact -> FreeMarkerTemplate [*];
351
352   context FreeMarkerTemplate inv:
353     nameExtension == "ftl";
354
355   context FreeMarkerTemplate inv:
356     refersTo.containsAll(reliesOnTemplate);
357
358   context FreeMarkerTemplate inv:
359     refersTo.containsAll(reliesOnJavaArtifact);
360
361   context JavaArtifact inv:
362     refersTo.containsAll(reliesOnTemplate);
363
364   // ____________________________________________________________
365   // Generator
366   // ____________________________________________________________
367
368   class Generator extends Tool {}
369
```

```
370  association uses [*] Generator -> FreeMarkerTemplate [*];
371
372  association /template
373    [*] Generator -> FreeMarkerTemplate [*];
374
375  association inputLanguage [*] Generator -> Language [*];
376
377  context Generator g, JavaArtifact a in g.consistsOf inv:
378    g.template == g.uses.addAll(g.template.reliesOnTemplate**)
379                        .addAll(a.reliesOnTemplate);
380
381  context Generator inv:
382    consistsOf.contains(template);
383
384  context Generator inv:
385    consistsOf.contains(template.reliesOnJavaArtifact);
386
387  // ____________________________________________________________
388  // Action
389  // ____________________________________________________________
390
391  class Action {
392    Date start;
393    Date end;
394  }
395
396  class Event extends Action {}
397
398  association subaction
399    [0..1] Action (parent) -> (sub) Action [*] <<ordered>>;
400
401  context Action inv:
402    start <= end;
403
404  context Event inv:
405    start == end;
406
407  context Action a, Action s in a.subaction inv:
408    a.start <= s.start && s.end <= a.end;
409
410  // ____________________________________________________________
411  // Execution Protocol
412  // ____________________________________________________________
```

```
413
414  class ExecutionProtocol {}
415
416  association ExecutionProtocol -> Action [*] <<ordered>>;
417
418  context ExecutionProtocol p inv:
419    p.action.nonEmpty implies (p.action.asSet ==
420      p.action[0].subaction**.add(action[0]));
421
422  context ExecutionProtocol p, int i,k inv:
423    (0 <= k && k <= p.action.size && 0 <= i && i <= k) implies
424      p.action[i].start <= p.action[k].start;
425
426  context ExecutionProtocol p, int i,k inv:
427    (0 <= k && k <= p.action.size && 0 <= i && i <= k) implies
428      p.action[i].end <= p.action[k].start;
429
430  // ____________________________________________________________
431  // ToolActions
432  // ____________________________________________________________
433
434  class ToolExecution           extends Action {}
435  class TemplateExecution       extends Action {}
436  class JavaMethodCall          extends Action {}
437  class ArtifactRead            extends Action {}
438  class ArtifactExistenceCheck  extends Action {
439    String fullName;
440  }
441  class ArtifactCreate          extends Action {}
442  class WriteString             extends Action {
443    String content;
444  }
445
446  association tool [*] ToolExecution -> Tool [1];
447
448  association writesTo
449    [*] TemplateExecution -> Artifact [0..1];
450
451  association template
452    [*] TemplateExecution -> FreeMarkerTemplate [1];
453
454  association file [*] JavaMethodCall -> JavaClassFile [1];
455
```

```
456   association reads [*] ArtifactRead -> Artifact [1];
457
458   association checks
459     [*] ArtifactExistenceCheck -> Artifact [0..1];
460
461   association creates [0..1] ArtifactCreate -> Artifact [1];
462
463   association writesTo [*] WriteString -> Artifact [1];
464
465   // ____________________________________________________________
466   // ToolExecution
467   // ____________________________________________________________
468
469   association /input [*] ToolExecution -> Artifact [*];
470   association /read [*] ToolExecution -> Artifact [*];
471   association /existenceCheck [*] ToolExecution -> Artifact [*];
472
473   context ToolExecution inv:
474     read == {a in ArtifactRead | a in subaction**}.reads;
475
476   context ToolExecution inv:
477     existenceCheck ==
478       {a in ArtifactExistenceCheck | a in subaction**}.checks;
479
480   context ToolExecution inv:
481     input == read.addAll(existenceCheck);
482
483   association /output [0..1] ToolExecution -> Artifact [*];
484
485   context ToolExecution inv:
486     output ==
487       {a in ArtifactCreate | a in subaction**}.creates;
488
489   association /creates [*] Artifact (source) -> Artifact [*];
490   association /existenceImpact
491     [*] Artifact (checked) -> Artifact [*];
492
493   context Artifact inv:
494     creates == read.output;
495
496   context Artifact inv:
497     existenceImpact == existenceCheck.output;
498
```

```
499   context Artifact inv:
500     produces.containsAll(creates);
501
502   context Artifact inv:
503     produces.containsAll(existenceImpact);
504
505   // _____________________________________________________________
506   // Action and Artifact Contribution
507   // _____________________________________________________________
508
509   association /contributesTo [*] Action -> Artifact [*];
510
511   context ToolExecution inv:
512     contributesTo == sub.contributesTo;
513
514   context ToolExecution inv:
515     contributesTo == output;
516
517   context TemplateExecution inv:
518     contributesTo == sub.contributesTo.add(writesTo);
519
520   context JavaMethodCall inv:
521     contributesTo == sub.contributesTo;
522
523   context ArtifactRead inv:
524     contributesTo == {};
525
526   context ArtifactExistenceCheck inv:
527     contributesTo == {};
528
529   context ArtifactCreate inv:
530     contributesTo == {creates};
531
532   context WriteString inv:
533     contributesTo == {writesTo};
534
535   association /contributionFT
536     [*] FreeMarkerTemplate (contributorFT) -> Artifact [*];
537
538   context FreeMarkerTemplate inv:
539     contributionFT == templateExecution.contributesTo;
540
541   context FreeMarkerTemplate inv:
```

```
542     produces.containsAll(contributionFT);
543
544   association /contributionJava
545     [*] JavaClassFile (contributorJava) -> Artifact [*];
546
547   context JavaClassFile inv:
548     contributionJava == javaMethodCall.contributesTo;
549
550   context JavaClassFile inv:
551     produces.containsAll(contributionJava);
552 }
```

Listing C.1: AM für MontiCore-basierte MDD-Projekte.

Anhang D.

Tagschema für artefaktbasierte Analysen

Die Tagtypen des in diesem Kapitel dargestellten Tagschemas können wie in Abschnitt 7.2.5 beschrieben zur Spezifikation der Nutzungsart von OCL-Ausdrücken verwendet werden.

TS

```
tagschema ArtifactBasedAnalysis {
  tagtype check : [analysis, manual]*
    for OCLInvariant;
  tagtype derive : [analysis, manual]*
    for OCLInvariant;
  tagtype metric : manual?
    for OCLInvariant;
}
```

Listing D.1: Tagschema für artefaktbasierte Analysen.

Anhang E.

MontiArT-Werkzeuge

MontiArT-Werkzeuge werden als ausführbare Jar-Archive zur Verfügung gestellt. Dabei können mehrere Werkzeuge in einem Jar-Archiv gebündelt werden. Tabelle E.1 enthält eine vollständige Liste aller mitgelieferten MontiArT-Werkzeuge und deren Verteilung auf ausführbaren Jar-Archive. Dabei wird in der Tabelle statt *artifacts-<Kurzname>-<Version>-cli.jar* nur der Kurzname für MontiArT-Archive verwendet.

Archiv-Kurzname	**OD-Version**	**Werkzeug**	**Abschnitt**
`artifactcontainer-extractor`	2.2.0	`ArtifactContainerExtractor`	8.2.2
`template-extractor`	2.2.0	`TemplateExtractor`	8.2.3
`java-extractor`	2.2.0	`JavaExtractor`	8.2.4
`language-tools`	0.2.2	`ODMerger`	8.3.1
		`CDODConsistencyChecker`	8.4.2
		`OD2JSON`	8.6.2
`ocl-generator`	–	`OCLGenerator`	8.4.3
`ocl-tools`	0.2.2	`ODVerifier`	8.4.4
		`ODAccumulator`	8.4.5
		`ODVerifier4A`	8.5.2
		`ODAccumulator4A`	8.5.2
`od-filter`	2.2.0	`ODFilter`	8.5.3
`analysis-tools`	0.2.2	`ArcReconstructor`	8.5.4
		`ACChecker`	8.5.4
`od-flattener`	2.2.0	`ODFlattener`	8.6.1
Realisiert als Shell-Skript.	–	`CDOCLSplitter`	8.6.3

Tabelle E.1.: Verteilung der mitgelieferten MontiArT-Werkzeuge auf die ausführbaren Archive.

Anhang F.

DEx-Artefaktanalyse

In diesem Kapitel sind ausgewählte, an der in Abschnitt 10.1 beschriebenen artefaktbasierten Analyse beteiligte Artefakte dargestellt.

F.1. Definition der DEx-MontiArt-Instanz als Skript

SH

```sh
#!/bin/bash
function error_exit
{
  echo "$(basename $0) $1: Tool exits with error code $2" 1>&2
  exit 1
}

if [[ $# -eq 2 ]]; then
  WORKSPACE=$1
  INPUTDATA=$2
else
  echo "Wrong Number of Arguments.
    Use ./toolchain.sh WORKSPACE INPUTDATA."
  exit 1
fi

echo 'The used workspace' $WORKSPACE
echo 'The used input data' $INPUTDATA

# resources folder
toolPath=$WORKSPACE/target/tools
modelPath=$WORKSPACE/src/main/resources/models

```

```
24 # input folder
25 inputGEN=$INPUTDATA/use/gen/dex
26 inputRTE=$INPUTDATA/rte/src
27 inputGTR=$INPUTDATA/gtr/src
28
29 #input artifacts
30 actionReport=$INPUTDATA/use/gen/reports/SocNet/17_ActionReport.od
31 dexdefinition=$modelPath/dexdefinition/DExDefinition.od
32
33 # output folder
34 resultsFolder=$WORKSPACE/target/analysis/DEx_dependencies
35 metricsFolder=$WORKSPACE/target/analysis/metrics
36
37 extractionFolder=$resultsFolder/01.extraction
38 mergedFolder=$resultsFolder/02.merged
39 accFolder=$resultsFolder/03.acc
40 scriptsFolder=$resultsFolder/04.scripts
41
42 tempFolder=$WORKSPACE/target/temp
43 containerTempFolder=$WORKSPACE/target/containerTemp
44
45 # results
46 artifactContainerOD=$extractionFolder/AritfactContainer.od
47 javaOD=$extractionFolder/Java.od
48 templateOD=$extractionFolder/Template.od
49 dexTrafoOD=$extractionFolder/DExTrafo.od
50 filteredActionReport=$extractionFolder/FilteredActionReport.od
51
52 flatArtifactContainerOD=$tempFolder/ArtifactContainer.od
53 flatJavaOD=$tempFolder/Java.od
54 filteredJavaOD=$tempFolder/FilteredJavaOD.od
55 flatTemplateOD=$tempFolder/Template.od
56 flatDExTrafoOD=$tempFolder/DExTrafo.od
57
58 transformedArtifactContainerOD=\
59 $tempFolder/ArtifactContainerTransformed.od
60 transformedJavaOD=$tempFolder/JavaTransformed.od
61 transformedTemplateOD=$tempFolder/TemplateTransformed.od
62 transformedDExTrafoOD=$tempFolder/DExTrafoTransformed.od
63 transformedActionReport=$tempFolder/ActionReportTransformed.od
64
65 mergedOD=$mergedFolder/Merged.od
66 moduleDefinition=$accFolder/a.models/ModuleDefinition.od
```

```
67 intendedArchitecture=$accFolder/a.models/IntendedArchitecture.od
68
69 # extractors
70 templateExtractor=$toolPath/templates-1.0.0-cli.jar
71 artifactContainerExtractor=\
72 $toolPath/artifactcontainer-1.0.0-cli.jar
73 javaExtractor=$toolPath/java-1.0.0-cli.jar
74 dexTrafoExtractor=$toolPath/dex-analysis-1.0.0-cli.jar
75
76 # tools
77 odFlattener=$toolPath/artifact-od-flattener-1.0.0-cli.jar
78 odFilter=$toolPath/artifact-od-filter-1.0.0-cli.jar
79 languageTool=$toolPath/artifact-language-tool-1.0.0-cli.jar
80 analysisTool=$toolPath/artifact-analysis-1.0.0-cli.jar
81
82 # options
83 equalsStrategy=fullName
84
85 echo "------------------------------"
86 echo "- Start extraction           -"
87 echo "------------------------------"
88
89 mkdir -p $tempFolder
90 mkdir -p $WORKSPACE/target/analysis/DEx_dependencies
91
92 echo 'extract containers'
93 java -jar $artifactContainerExtractor -r $INPUTDATA \
94   "$inputGEN;$inputRTE;$inputGTR" $artifactContainerOD \
95   $tempFolder || error_exit $LINENO $?
96 echo 'extract containers finished'
97
98 echo 'extract java'
99 java -jar $javaExtractor -r $INPUTDATA \
100   "$inputGEN;$inputRTE;$inputGTR" $javaOD || error_exit $LINENO $?
101 echo 'extract java finished'
102
103 echo 'extract templates'
104 java -jar $templateExtractor -r $INPUTDATA \
105   $inputGTR $templateOD || error_exit $LINENO $?
106 echo 'extract templates finished'
107
108 echo 'extract trafos'
109 java -jar $dexTrafoExtractor -r $INPUTDATA \
```

```
110   $inputGTR $dexTrafoOD || error_exit $LINENO $?
111 echo 'extract trafos finished'
112
113 echo "------------------------------"
114 echo "- filter Action Report       -"
115 echo "------------------------------"
116
117 java -jar $odFilter -o "FreeMarkerTemplate,JavaSourceFile" \
118   $actionReport $filteredActionReport || error_exit $LINENO $?
119
120 echo "------------------------------"
121 echo "- create flat ODs            -"
122 echo "------------------------------"
123 java -jar $odFlattener $javaOD $flatJavaOD \
124   || error_exit $LINENO $?
125 java -jar $odFlattener $templateOD $flatTemplateOD \
126   || error_exit $LINENO $?
127 java -jar $odFlattener $artifactContainerOD \
128   $flatArtifactContainerOD || error_exit $LINENO $?
129 java -jar $odFlattener $dexTrafoOD $flatDExTrafoOD \
130   || error_exit $LINENO $?
131
132 echo "------------------------------"
133 echo "- filter flat JavaOD         -"
134 echo "------------------------------"
135 java -jar $odFilter -o "JavaSourceFile" $flatJavaOD \
136   $filteredJavaOD || error_exit $LINENO $?
137
138 echo "------------------------------"
139 echo "- transform flat ODs         -"
140 echo "- to old OD version          -"
141 echo "------------------------------"
142
143 cat $filteredJavaOD | sed 's/};/}/g' > $transformedJavaOD
144 cat $flatDExTrafoOD | sed 's/};/}/g' > $transformedDExTrafoOD
145 cat $flatArtifactContainerOD |
146 sed -e 's/};/}/g' \
147     -e '/modified/d' > $transformedArtifactContainerOD
148 cat $flatTemplateOD | sed 's/};/}/g' > $transformedTemplateOD
149 cat $filteredActionReport | sed 's/};/}/g' \
150   > $transformedActionReport
151
152 echo "------------------------------"
```

```
153 echo "- Merge transformed ODs      -"
154 echo "------------------------------"
155
156 java -jar $languageTool merger "$transformedTemplateOD;\
157 $transformedJavaOD;$transformedArtifactContainerOD;\
158 $transformedDExTrafoOD;$transformedActionReport" \
159   $mergedOD $equalsStrategy || error_exit $LINENO $?
160
161 echo "------------------------------"
162 echo "- Moduledefinition           -"
163 echo "------------------------------"
164
165 mkdir -p $accFolder/a.models || error_exit $LINENO $?
166 mkdir -p $accFolder/b.mergedmodule || error_exit $LINENO $?
167 mkdir -p $accFolder/c.modulerelations || error_exit $LINENO $?
168
169 java -jar $odFilter -l "artifact" $dexdefinition \
170   $moduleDefinition || error_exit $LINENO $?
171 sed -i 's/};/}/g' $moduleDefinition
172
173 echo "------------------------------"
174 echo "- Intended Architecture       -"
175 echo "------------------------------"
176
177 java -jar $odFilter -o "Module" $dexdefinition \
178   $intendedArchitecture || error_exit $LINENO $?
179 sed -i 's/};/}/g' $intendedArchitecture
180
181 echo "------------------------------"
182 echo "- ACC Analyse                -"
183 echo "------------------------------"
184
185 java -jar $languageTool merger  "$moduleDefinition;$mergedOD" \
186   $accFolder/b.mergedmodule/MergedModule.od $equalsStrategy \
187   || error_exit $LINENO $?
188
189 java -jar $analysisTool Modules \
190   $accFolder/b.mergedmodule/MergedModule.od \
191   $accFolder/c.modulerelations/ModuleRelations.od \
192   || error_exit $LINENO $?
193
194 java -jar $analysisTool ArcDiff $intendedArchitecture \
195   $accFolder/c.modulerelations/ModuleRelations.od \
```

```
196   $accFolder/c.modulerelations/DifferenceArchitecture.od \
197   || error_exit $LINENO $?
198
199 echo "------------------------------"
200 echo "- Finished                   -"
201 echo "------------------------------"
```

Listing F.1: Shell-Skript zur Definition einer ausführbarenMontiArT-Instanz.

F.2. Moduldefinition und Soll-Architektur

OD

```
objectdiagram Definition {

// GTR GUI
  @gtr_gui_module:Module {
    simpleName = "gtr_gui";
  };

  @gtr_cd2swing:Directory {
    isRoot = false;
    fullName = "gtr/src/cd2swing";
    simpleName = "cd2swing";
  };

  link artifact @gtr_gui_module -> @gtr_cd2swing;

// GTR DATA
  @gtr_data_module:Module {
    simpleName = "gtr_data";
  };

  @gtr_cd2data:Directory {
    simpleName = "cd2data";
    fullName = "gtr/src/cd2data";
    isRoot = false;
  };

  link artifact @gtr_data_module -> @gtr_cd2data;

// GTR PERSISTENCE
  @gtr_persistence_module:Module {
    simpleName = "gtr_persistence";
  };

  @gtr_cd2persistence:Directory {
    simpleName = "cd2persistence";
    fullName = "gtr/src/cd2persistence";
    isRoot = false;
  };

```

```
40   link artifact @gtr_persistence_module ->
41     @gtr_cd2persistence;
42
43 // GTR COMMON
44   @gtr_common_module:Module {
45     simpleName = "gtr_common";
46   };
47
48   @gtr_common:Directory {
49     simpleName = "common";
50     fullName = "gtr/src/common";
51     isRoot = false;
52   };
53
54   link artifact @gtr_common_module -> @gtr_common;
55
56 // RTE GUI
57   @rte_gui_module:Module {
58       simpleName = "rte_gui";
59   };
60
61   @rte_gui:Directory {
62     simpleName = "gui";
63     fullName = "rte/src/dex/gui";
64     isRoot = false;
65   };
66
67   @rte_search:Directory {
68     simpleName = "search";
69     fullName = "rte/src/dex/search";
70     isRoot = false;
71   };
72
73   @rte_help:Directory {
74     simpleName = "help";
75     fullName = "rte/src/dex/help";
76     isRoot = false;
77   };
78
79   link artifact @rte_gui_module -> @rte_gui;
80   link artifact @rte_gui_module -> @rte_search;
81   link artifact @rte_gui_module -> @rte_help;
82
```

```
// RTE DATA
  @rte_data_module:Module {
    simpleName = "rte_data";
  };

  @rte_data:Directory {
    simpleName = "data";
    fullName = "rte/src/dex/data";
    isRoot = false;
  };

  link artifact @rte_data_module -> @rte_data;

// RTE PERSISTENCE
  @rte_persistence_module:Module {
      simpleName = "rte_persistence";
    };

  @rte_persistence:Directory {
    simpleName = "persistence";
    fullName = "rte/src/dex/persistence";
    isRoot = false;
  };

   @rte_serveraccess:Directory {
    simpleName = "access";
    fullName = "rte/src/dex/server/access";
    isRoot = false;
  };

   link artifact @rte_persistence_module -> @rte_persistence;
   link artifact @rte_persistence_module -> @rte_serveraccess;

// RTE UTIL
  @rte_util_module:Module {
    simpleName = "rte_util";
  };

  @rte_util:Directory {
    simpleName = "util";
    fullName = "rte/src/util";
    isRoot = false;
  };
```

```
126
127     link artifact @rte_util_module -> @rte_util ;
128
129 // GEN GUI
130   @gen_gui_module:Module {
131     simpleName = "gen_gui";
132   };
133
134   @gen_gui:Directory {
135     simpleName = "gui";
136     fullName = "use/gen/dex/socnet/gui";
137     isRoot = false;
138   };
139
140   link artifact @gen_gui_module -> @gen_gui ;
141
142 // GEN DATA
143   @gen_data_module:Module{
144     simpleName = "gen_data";
145   };
146
147   @Group!java:JavaSourceFile{
148     simpleName = "Group";
149     fullName = "use/gen/dex/socnet/Group.java";
150   };
151
152   @GroupBuilder!java:JavaSourceFile{
153     simpleName = "GroupBuilder";
154     fullName = "use/gen/dex/socnet/GroupBuilder.java";
155   };
156
157   @GroupImpl!java:JavaSourceFile{
158     simpleName = "GroupImpl";
159     fullName = "use/gen/dex/socnet/GroupImpl.java";
160   };
161
162   @GroupManager!java:JavaSourceFile{
163     simpleName = "GroupManager";
164     fullName = "use/gen/dex/socnet/GroupManager.java";
165   };
166
167   @GroupProxy!java:JavaSourceFile{
168     simpleName = "GroupProxy";
```

```
169     fullName = "use/gen/dex/socnet/GroupProxy.java";
170   };
171
172   @InstantMessage!java:JavaSourceFile{
173     simpleName = "InstantMessage";
174     fullName = "use/gen/dex/socnet/InstantMessage.java";
175   };
176
177   @InstantMessageBuilder!java:JavaSourceFile{
178     simpleName = "InstantMessageBuilder";
179     fullName = "use/gen/dex/socnet/InstantMessageBuilder.java";
180   };
181
182   @InstantMessageImpl!java:JavaSourceFile{
183     simpleName = "InstantMessageImpl";
184     fullName = "use/gen/dex/socnet/InstantMessageImpl.java";
185   };
186
187   @InstantMessageManager!java:JavaSourceFile{
188     simpleName = "InstantMessageManager";
189     fullName = "use/gen/dex/socnet/InstantMessageManager.java";
190   };
191
192   @InstantMessageProxy!java:JavaSourceFile{
193     simpleName = "InstantMessageProxy";
194     fullName = "use/gen/dex/socnet/InstantMessageProxy.java";
195   };
196
197   @Main!java:JavaSourceFile{
198     simpleName = "Main";
199     fullName = "use/gen/dex/socnet/Main.java";
200   };
201
202   @Person!java:JavaSourceFile{
203     simpleName = "Person";
204     fullName = "use/gen/dex/socnet/Person.java";
205   };
206
207   @PersonBuilder!java:JavaSourceFile{
208     simpleName = "PersonBuilder";
209     fullName = "use/gen/dex/socnet/PersonBuilder.java";
210   };
211
```

```
212  @PersonImpl!java:JavaSourceFile{
213    simpleName = "PersonImpl";
214    fullName = "use/gen/dex/socnet/PersonImpl.java";
215  };
216
217  @PersonManager!java:JavaSourceFile{
218    simpleName = "PersonManager";
219    fullName = "use/gen/dex/socnet/PersonManager.java";
220  };
221
222  @PersonProxy!java:JavaSourceFile{
223    simpleName = "PersonProxy";
224    fullName = "use/gen/dex/socnet/PersonProxy.java";
225  };
226
227  @Photo!java:JavaSourceFile{
228    simpleName = "Photo";
229    fullName = "use/gen/dex/socnet/Photo.java";
230  };
231
232  @PhotoBuilder!java:JavaSourceFile{
233    simpleName = "PhotoBuilder";
234    fullName = "use/gen/dex/socnet/PhotoBuilder.java";
235  };
236
237  @PhotoImpl!java:JavaSourceFile{
238    simpleName = "PhotoImpl";
239    fullName = "use/gen/dex/socnet/PhotoImpl.java";
240  };
241
242  @PhotoManager!java:JavaSourceFile{
243    simpleName = "PhotoManager";
244    fullName = "use/gen/dex/socnet/PhotoManager.java";
245  };
246
247  @PhotoMessage!java:JavaSourceFile{
248    simpleName = "PhotoMessage";
249    fullName = "use/gen/dex/socnet/PhotoMessage.java";
250  };
251
252  @PhotoMessageBuilder!java:JavaSourceFile{
253    simpleName = "PhotoMessageBuilder";
254    fullName = "use/gen/dex/socnet/PhotoMessageBuilder.java";
```

```
255   };
256
257   @PhotoMessageImpl!java:JavaSourceFile{
258     simpleName = "PhotoMessageImpl";
259     fullName = "use/gen/dex/socnet/PhotoMessageImpl.java";
260   };
261
262   @PhotoMessageManager!java:JavaSourceFile{
263     simpleName = "PhotoMessageManager";
264     fullName = "use/gen/dex/socnet/PhotoMessageManager.java";
265   };
266
267   @PhotoMessageProxy!java:JavaSourceFile{
268     simpleName = "PhotoMessageProxy";
269     fullName = "use/gen/dex/socnet/PhotoMessageProxy.java";
270   };
271
272   @PhotoProxy!java:JavaSourceFile{
273     simpleName = "PhotoProxy";
274     fullName = "use/gen/dex/socnet/PhotoProxy.java";
275   };
276
277   @Post!java:JavaSourceFile{
278     simpleName = "Post";
279     fullName = "use/gen/dex/socnet/Post.java";
280   };
281
282   @PostManager!java:JavaSourceFile{
283     simpleName = "PostManager";
284     fullName = "use/gen/dex/socnet/PostManager.java";
285   };
286
287   @Profile!java:JavaSourceFile{
288     simpleName = "Profile";
289     fullName = "use/gen/dex/socnet/Profile.java";
290   };
291
292   @ProfileBuilder!java:JavaSourceFile{
293     simpleName = "ProfileBuilder";
294     fullName = "use/gen/dex/socnet/ProfileBuilder.java";
295   };
296
297   @ProfileImpl!java:JavaSourceFile{
```

```
298     simpleName = "ProfileImpl";
299     fullName = "use/gen/dex/socnet/ProfileImpl.java";
300   };
301
302   @ProfileManager!java:JavaSourceFile{
303     simpleName = "ProfileManager";
304     fullName = "use/gen/dex/socnet/ProfileManager.java";
305   };
306
307   @ProfileProxy!java:JavaSourceFile{
308     simpleName = "ProfileProxy";
309     fullName = "use/gen/dex/socnet/ProfileProxy.java";
310   };
311
312   @Relationship!java:JavaSourceFile{
313     simpleName = "Relationship";
314     fullName = "use/gen/dex/socnet/Relationship.java";
315   };
316
317   @RelationshipBuilder!java:JavaSourceFile{
318     simpleName = "RelationshipBuilder";
319     fullName = "use/gen/dex/socnet/RelationshipBuilder.java";
320   };
321
322   @RelationshipImpl!java:JavaSourceFile{
323     simpleName = "RelationshipImpl";
324     fullName = "use/gen/dex/socnet/RelationshipImpl.java";
325   };
326
327   @RelationshipManager!java:JavaSourceFile{
328     simpleName = "RelationshipManager";
329     fullName = "use/gen/dex/socnet/RelationshipManager.java";
330   };
331
332   @RelationshipProxy!java:JavaSourceFile{
333     simpleName = "RelationshipProxy";
334     fullName = "use/gen/dex/socnet/RelationshipProxy.java";
335   };
336
337   @RelationType!java:JavaSourceFile{
338     simpleName = "RelationType";
339     fullName = "use/gen/dex/socnet/RelationType.java";
340   };
```

```
341
342   @Tag!java:JavaSourceFile{
343     simpleName = "Tag";
344     fullName = "use/gen/dex/socnet/Tag.java";
345   };
346
347   @TagBuilder!java:JavaSourceFile{
348     simpleName = "TagBuilder";
349     fullName = "use/gen/dex/socnet/TagBuilder.java";
350   };
351
352   @TagImpl!java:JavaSourceFile{
353     simpleName = "TagImpl";
354     fullName = "use/gen/dex/socnet/TagImpl.java";
355   };
356
357   @TagManager!java:JavaSourceFile{
358     simpleName = "TagManager";
359     fullName = "use/gen/dex/socnet/TagManager.java";
360   };
361
362   @TagProxy!java:JavaSourceFile{
363     simpleName = "TagProxy";
364     fullName = "use/gen/dex/socnet/TagProxy.java";
365   };
366
367   link artifact @gen_data_module -> @Group!java;
368   link artifact @gen_data_module -> @GroupBuilder!java;
369   link artifact @gen_data_module -> @GroupImpl!java;
370   link artifact @gen_data_module -> @GroupManager!java;
371   link artifact @gen_data_module -> @GroupProxy!java;
372   link artifact @gen_data_module -> @InstantMessage!java;
373   link artifact @gen_data_module ->
374     @InstantMessageBuilder!java;
375   link artifact @gen_data_module -> @InstantMessageImpl!java;
376   link artifact @gen_data_module ->
377     @InstantMessageManager!java;
378   link artifact @gen_data_module -> @InstantMessageProxy!java;
379   link artifact @gen_data_module -> @Main!java;
380   link artifact @gen_data_module -> @Person!java;
381   link artifact @gen_data_module -> @PersonBuilder!java;
382   link artifact @gen_data_module -> @PersonImpl!java;
383   link artifact @gen_data_module -> @PersonManager!java;
```

```
384   link artifact @gen_data_module -> @PersonProxy!java;
385   link artifact @gen_data_module -> @Photo!java;
386   link artifact @gen_data_module -> @PhotoBuilder!java;
387   link artifact @gen_data_module -> @PhotoImpl!java;
388   link artifact @gen_data_module -> @PhotoManager!java;
389   link artifact @gen_data_module -> @PhotoMessage!java;
390   link artifact @gen_data_module -> @PhotoMessageBuilder!java;
391   link artifact @gen_data_module -> @PhotoMessageImpl!java;
392   link artifact @gen_data_module -> @PhotoMessageManager!java;
393   link artifact @gen_data_module -> @PhotoMessageProxy!java;
394   link artifact @gen_data_module -> @PhotoProxy!java;
395   link artifact @gen_data_module -> @Post!java;
396   link artifact @gen_data_module -> @PostManager!java;
397   link artifact @gen_data_module -> @Profile!java;
398   link artifact @gen_data_module -> @ProfileBuilder!java;
399   link artifact @gen_data_module -> @ProfileImpl!java;
400   link artifact @gen_data_module -> @ProfileManager!java;
401   link artifact @gen_data_module -> @ProfileProxy!java;
402   link artifact @gen_data_module -> @Relationship!java;
403   link artifact @gen_data_module -> @RelationshipBuilder!java;
404   link artifact @gen_data_module -> @RelationshipImpl!java;
405   link artifact @gen_data_module -> @RelationshipManager!java;
406   link artifact @gen_data_module -> @RelationshipProxy!java;
407   link artifact @gen_data_module -> @RelationType!java;
408   link artifact @gen_data_module -> @Tag!java;
409   link artifact @gen_data_module -> @TagBuilder!java;
410   link artifact @gen_data_module -> @TagImpl!java;
411   link artifact @gen_data_module -> @TagManager!java;
412   link artifact @gen_data_module -> @TagProxy!java;
413
414 // GEN PERSISTENCE
415   @gen_persistence_module:Module {
416     simpleName = "gen_persistence";
417   };
418
419   @gen_persistence:Directory {
420     simpleName = "persistence";
421     fullName = "use/gen/dex/socnet/persistence";
422     isRoot = false;
423   };
424
425   link artifact @gen_persistence_module ->
426     @gen_persistence ;
```

```
427
428 // DESIRED RELATIONS
429   link intendedReference @gtr_gui_module ->
430     @gtr_data_module;
431   link intendedReference @gtr_gui_module ->
432     @gtr_common_module;
433   link intendedContribution @gtr_gui_module ->
434     @gen_gui_module;
435   link intendedContribution @gtr_gui_module ->
436     @gen_data_module;
437
438   link intendedReference @gtr_data_module ->
439     @gtr_persistence_module;
440   link intendedReference @gtr_data_module ->
441     @gtr_common_module;
442   link intendedContribution @gtr_data_module ->
443     @gen_data_module;
444
445   link intendedReference @gtr_persistence_module ->
446     @gtr_common_module;
447   link intendedContribution @gtr_persistence_module ->
448     @gen_data_module;
449   link intendedContribution @gtr_persistence_module ->
450     @gen_persistence_module;
451
452   link intendedReference @gen_gui_module -> @rte_gui_module;
453   link intendedReference @gen_gui_module -> @rte_data_module;
454   link intendedReference @gen_gui_module -> @gen_data_module;
455
456   link intendedReference @gen_data_module -> @rte_data_module;
457   link intendedReference @gen_data_module ->
458     @gen_persistence_module;
459
460   link intendedReference @gen_persistence_module ->
461     @gen_data_module;
462   link intendedReference @gen_persistence_module ->
463     @rte_data_module;
464   link intendedReference @gen_persistence_module ->
465     @rte_persistence_module;
466
467   link intendedReference @rte_gui_module -> @rte_data_module;
468   link intendedReference @rte_gui_module -> @rte_util_module;
469
```

```
470   link intendedReference @rte_data_module ->
471     @rte_persistence_module;
472   link intendedReference @rte_data_module -> @rte_util_module;
473
474   link intendedReference @rte_persistence_module ->
475     @rte_util_module;
476 }
```

Listing F.2: Kombination aus Moduldefinition und der Soll-Architektur für das DEx-Projekts.

F.3. Ist-Architektur

OD
«IstArc»

```
objectdiagram ModuleRelations {

  @rte_data_module:Module {
    simpleName = "rte_data";
  };

  @gtr_data_module:Module {
    simpleName = "gtr_data";
  };

  @rte_persistence_module:Module {
    simpleName = "rte_persistence";
  };

  @gen_persistence_module:Module {
    simpleName = "gen_persistence";
  };

  @gen_gui_module:Module {
    simpleName = "gen_gui";
  };

  @gtr_common_module:Module {
    simpleName = "gtr_common";
  };

  @gen_data_module:Module {
    simpleName = "gen_data";
  };

  @gtr_gui_module:Module {
    simpleName = "gtr_gui";
  };

  @gtr_persistence_module:Module {
    simpleName = "gtr_persistence";
  };

  @rte_util_module:Module {
```

```
40      simpleName = "rte_util";
41    };
42
43    @rte_gui_module:Module {
44      simpleName = "rte_gui";
45    };
46
47    link reliesOnModule @rte_data_module ->
48      @rte_persistence_module;
49    link reliesOnModule @gtr_data_module ->
50      @gtr_common_module;
51    link contributionModule @gtr_data_module ->
52      @gen_data_module;
53    link contributionModule @gtr_data_module ->
54      @gen_persistence_module;
55    link contributionModule @gtr_data_module ->
56      @gen_gui_module;
57    link reliesOnModule @rte_persistence_module ->
58      @rte_gui_module;
59    link reliesOnModule @gen_persistence_module ->
60      @rte_data_module;
61    link reliesOnModule @gen_persistence_module ->
62      @rte_persistence_module;
63    link reliesOnModule @gen_persistence_module ->
64      @gen_data_module;
65    link reliesOnModule @gen_persistence_module ->
66      @rte_util_module;
67    link reliesOnModule @gen_persistence_module ->
68      @rte_gui_module;
69    link reliesOnModule @gen_gui_module -> @rte_gui_module;
70    link reliesOnModule @gen_gui_module ->
71      @rte_persistence_module;
72    link reliesOnModule @gen_gui_module -> @gen_data_module;
73    link reliesOnModule @gen_gui_module -> @rte_data_module;
74    link reliesOnModule @gen_gui_module ->
75      @gen_persistence_module;
76    link reliesOnModule @gen_gui_module -> @rte_util_module;
77    link contributionModule @gtr_common_module ->
78      @gen_persistence_module;
79    link contributionModule @gtr_common_module ->
80      @gen_data_module;
81    link contributionModule @gtr_common_module ->
82      @gen_gui_module;
```

```
83    link reliesOnModule @gen_data_module -> @rte_data_module;
84    link reliesOnModule @gen_data_module ->
85      @rte_persistence_module;
86    link reliesOnModule @gen_data_module ->
87      @gen_persistence_module;
88    link reliesOnModule @gen_data_module -> @rte_gui_module;
89    link reliesOnModule @gen_data_module -> @gen_gui_module;
90    link reliesOnModule @gtr_gui_module -> @gtr_data_module;
91    link reliesOnModule @gtr_gui_module -> @gtr_common_module;
92    link contributionModule @gtr_gui_module -> @gen_gui_module;
93    link contributionModule @gtr_gui_module ->
94      @gen_data_module;
95    link reliesOnModule @gtr_persistence_module ->
96      @gtr_common_module;
97    link contributionModule @gtr_persistence_module ->
98      @gen_persistence_module;
99    link reliesOnModule @gtr_persistence_module ->
100     @gtr_data_module;
101   link contributionModule @gtr_persistence_module ->
102     @gen_data_module;
103   link reliesOnModule @rte_util_module -> @rte_gui_module;
104   link reliesOnModule @rte_gui_module -> @rte_data_module;
105   link reliesOnModule @rte_gui_module -> @rte_util_module;
106   link reliesOnModule @rte_gui_module ->
107     @rte_persistence_module;
108 }
```

Listing F.3: Berechnete Ist-Architektur des DEx-Projekts.

F.4. Differenzarchitektur

OD
«DiffArc»

```
objectdiagram DifferenceArchitecture {

  @rte_data_module:Module {
    simpleName = "rte_data";
  };

  @gtr_data_module:Module {
    simpleName = "gtr_data";
  };

  @gen_persistence_module:Module {
    simpleName = "gen_persistence";
  };

  @rte_persistence_module:Module {
    simpleName = "rte_persistence";
  };

  @gen_gui_module:Module {
    simpleName = "gen_gui";
  };

  @gtr_common_module:Module {
    simpleName = "gtr_common";
  };

  @gen_data_module:Module {
    simpleName = "gen_data";
  };

  @gtr_persistence_module:Module {
    simpleName = "gtr_persistence";
  };

  @rte_util_module:Module {
    simpleName = "rte_util";
  };

  @rte_gui_module:Module {
```

```
40      simpleName = "rte_gui";
41    };
42
43    link unintendedContribution @gtr_data_module ->
44      @gen_persistence_module;
45    link unintendedContribution @gtr_data_module ->
46      @gen_gui_module;
47    link unintendedReference @rte_persistence_module ->
48      @rte_gui_module;
49    link unintendedReference @gen_persistence_module ->
50      @rte_util_module;
51    link unintendedReference @gen_persistence_module ->
52      @rte_gui_module;
53    link unintendedReference @gen_gui_module ->
54      @rte_persistence_module;
55    link unintendedReference @gen_gui_module ->
56      @gen_persistence_module;
57    link unintendedReference @gen_gui_module ->
58      @rte_util_module;
59    link unintendedContribution @gtr_common_module ->
60      @gen_persistence_module;
61    link unintendedContribution @gtr_common_module ->
62      @gen_data_module;
63    link unintendedContribution @gtr_common_module ->
64      @gen_gui_module;
65    link unintendedReference @gen_data_module ->
66      @rte_persistence_module;
67    link unintendedReference @gen_data_module ->
68      @rte_gui_module;
69    link unintendedReference @gen_data_module ->
70      @gen_gui_module;
71    link unintendedReference @gtr_persistence_module ->
72      @gtr_data_module;
73    link unintendedReference @rte_util_module ->
74      @rte_gui_module;
75    link unintendedReference @rte_gui_module ->
76      @rte_persistence_module;
77    link missingReference @gtr_data_module ->
78      @gtr_persistence_module;
79    link missingReference @rte_data_module ->
80      @rte_util_module;
81    link missingReference @rte_persistence_module ->
82      @rte_util_module;
```

```
83 }
```

Listing F.4: Berechnete Differenzarchitektur des DEx-Projekts.

Anhang G.

BumperBot-Artefaktanalyse

In diesem Kapitel sind ausgewählte, an der in Abschnitt 10.2 beschriebenen artefaktbasierten Analyse beteiligte Artefakte dargestellt.

G.1. Extrahierte Artefaktdaten

OD
«ArtData»

```
objectdiagram MontiArcAutomatonArtifacts {

  @Character!java:JavaSourceFile{
    simpleName = "Character";
    fullName = "java/lang/Character.java";
  };

  @Object!java:JavaSourceFile{
    simpleName = "Object";
    fullName = "java/lang/Object.java";
  };

  @Timer!maa:MAAModelFile{
    simpleName = "Timer";
    fullName = "de/montiarcautomaton/lejos/lib/timer/Timer.maa";
  };

  @Motor!maa:MAAModelFile{
    simpleName = "Motor";
    fullName = "de/montiarcautomaton/lejos/lib/motor/Motor.maa";
  };

  @BumpControl!maa:MAAModelFile{
```

```
24      simpleName = "BumpControl";
25      fullName = "bumperbot/BumpControl.maa";
26    };
27
28    @Optional!java:JavaSourceFile{
29      simpleName = "Optional";
30      fullName = "java/util/Optional.java";
31    };
32
33    @Datatypes!cd(!3):CDModelFile{
34      simpleName = "Datatypes";
35      fullName = "de/montiarcautomaton/lejos/lib/Datatypes.cd";
36    };
37
38    @Comparable!java:JavaSourceFile{
39      simpleName = "Comparable";
40      fullName = "java/lang/Comparable.java";
41    };
42
43    @Number!java:JavaSourceFile{
44      simpleName = "Number";
45      fullName = "java/lang/Number.java";
46    };
47
48    @Datatypes!cd:CDModelFile{
49      simpleName = "Datatypes";
50      fullName = "speedbumper/Datatypes.cd";
51    };
52
53    @Logger!maa:MAAModelFile{
54      simpleName = "Logger";
55      fullName = "de/montiarcautomaton/lejos/lib/logger/Logger.maa";
56    };
57
58    @List!java:JavaSourceFile{
59      simpleName = "List";
60      fullName = "java/util/List.java";
61    };
62
63    @Datatypes!cd(!2):CDModelFile{
64      simpleName = "Datatypes";
65      fullName = "de/montiarcautomaton/lejos/lib/timer/
66        Datatypes.cd";
```

```
67  };
68
69  @Integer!java:JavaSourceFile{
70    simpleName = "Integer";
71    fullName = "java/lang/Integer.java";
72  };
73
74  @String!java:JavaSourceFile{
75    simpleName = "String";
76    fullName = "java/lang/String.java";
77  };
78
79  @Set!java:JavaSourceFile{
80    simpleName = "Set";
81    fullName = "java/util/Set.java";
82  };
83
84  @BumpControl!maa(!2):MAAModelFile{
85    simpleName = "BumpControl";
86    fullName = "speedbumper/BumpControl.maa";
87  };
88
89  @BumperBot!maa(!2):MAAModelFile{
90    simpleName = "BumperBot";
91    fullName = "bumperbot/BumperBot.maa";
92  };
93
94  @Ultrasonic!maa:MAAModelFile{
95    simpleName = "Ultrasonic";
96    fullName = "de/montiarcautomaton/lejos/lib/ultrasonic/
97      Ultrasonic.maa";
98  };
99
100 @Datatypes!cd(!4):CDModelFile{
101   simpleName = "Datatypes";
102   fullName = "de/montiarcautomaton/lejos/lib/motor/
103     Datatypes.cd";
104 };
105
106 @Double!java:JavaSourceFile{
107   simpleName = "Double";
108   fullName = "java/lang/Double.java";
109 };
```

```
110
111   @BumperBot!maa:MAAModelFile{
112     simpleName = "BumperBot";
113     fullName = "speedbumper/BumperBot.maa";
114   };
115
116   @Boolean!java:JavaSourceFile{
117     simpleName = "Boolean";
118     fullName = "java/lang/Boolean.java";
119   };
120
121   @BumpSpeed!maa:MAAModelFile{
122     simpleName = "BumpSpeed";
123     fullName = "speedbumper/BumpSpeed.maa";
124   };
125
126   link imports @Timer!maa -> @Datatypes!cd(!2);
127   link imports @Motor!maa -> @Datatypes!cd(!3);
128   link imports @Motor!maa -> @Datatypes!cd(!4);
129   link imports @BumpControl!maa -> @Datatypes!cd(!4);
130   link imports @BumpControl!maa -> @Datatypes!cd(!2);
131   link imports @BumpControl!maa(!2) -> @Datatypes!cd(!4);
132   link imports @BumpControl!maa(!2) -> @Datatypes!cd(!2);
133   link imports @BumpControl!maa(!2) -> @Datatypes!cd;
134   link imports @BumperBot!maa(!2) -> @Datatypes!cd(!3);
135   link imports @BumperBot!maa(!2) -> @Timer!maa;
136   link imports @BumperBot!maa(!2) -> @Datatypes!cd(!2);
137   link imports @BumperBot!maa(!2) -> @Motor!maa;
138   link imports @BumperBot!maa(!2) -> @Datatypes!cd(!4);
139   link imports @BumperBot!maa(!2) -> @Ultrasonic!maa;
140   link imports @BumperBot!maa(!2) -> @Logger!maa;
141   link imports @Ultrasonic!maa -> @Datatypes!cd(!3);
142   link imports @BumperBot!maa -> @Datatypes!cd(!3);
143   link imports @BumperBot!maa -> @Timer!maa;
144   link imports @BumperBot!maa -> @Datatypes!cd(!2);
145   link imports @BumperBot!maa -> @Motor!maa;
146   link imports @BumperBot!maa -> @Datatypes!cd(!4);
147   link imports @BumperBot!maa -> @Ultrasonic!maa;
148   link imports @BumperBot!maa -> @Logger!maa;
149   link imports @BumpSpeed!maa -> @Datatypes!cd;
150   link typeUsage @Timer!maa -> @Datatypes!cd(!2);
151   link typeUsage @Timer!maa -> @Integer!java;
152   link typeUsage @Motor!maa -> @Datatypes!cd(!3);
```

```
153   link typeUsage @Motor!maa -> @Datatypes!cd(!4);
154   link typeUsage @Motor!maa -> @Integer!java;
155   link typeUsage @BumpControl!maa -> @Datatypes!cd(!2);
156   link typeUsage @BumpControl!maa -> @String!java;
157   link typeUsage @BumpControl!maa -> @Integer!java;
158   link typeUsage @BumpControl!maa -> @Datatypes!cd(!4);
159   link typeUsage @Logger!maa -> @String!java;
160   link typeUsage @BumpControl!maa(!2) -> @Datatypes!cd(!2);
161   link typeUsage @BumpControl!maa(!2) -> @String!java;
162   link typeUsage @BumpControl!maa(!2) -> @Integer!java;
163   link typeUsage @BumpControl!maa(!2) -> @Datatypes!cd(!4);
164   link typeUsage @BumpControl!maa(!2) -> @Datatypes!cd;
165   link typeUsage @BumperBot!maa(!2) -> @Timer!maa;
166   link typeUsage @BumperBot!maa(!2) -> @Motor!maa;
167   link typeUsage @BumperBot!maa(!2) -> @Datatypes!cd(!3);
168   link typeUsage @BumperBot!maa(!2) -> @Ultrasonic!maa;
169   link typeUsage @BumperBot!maa(!2) -> @BumpControl!maa;
170   link typeUsage @BumperBot!maa(!2) -> @Logger!maa;
171   link typeUsage @Ultrasonic!maa -> @Datatypes!cd(!3);
172   link typeUsage @Ultrasonic!maa -> @Integer!java;
173   link typeUsage @BumperBot!maa -> @Timer!maa;
174   link typeUsage @BumperBot!maa -> @Motor!maa;
175   link typeUsage @BumperBot!maa -> @Datatypes!cd(!3);
176   link typeUsage @BumperBot!maa -> @BumpSpeed!maa;
177   link typeUsage @BumperBot!maa -> @Ultrasonic!maa;
178   link typeUsage @BumperBot!maa -> @BumpControl!maa(!2);
179   link typeUsage @BumperBot!maa -> @Logger!maa;
180   link typeUsage @BumpSpeed!maa -> @Integer!java;
181   link typeUsage @BumpSpeed!maa -> @Datatypes!cd;
182 }
```

Listing G.1: Extrahierte Artefaktdaten für die Eingabeartefakte beider Varianten des BumperBots.

G.2. Ungenutzte Importe

OD
«ArtData»

```
objectdiagram BumperBotAnalysis {

  @BumperBot!maa(!2):MAAModelFile{
    simpleName = "BumperBot";
    fullName = "bumperbot/BumperBot.maa";
  };

  @Datatypes!cd(!2):CDModelFile{
    simpleName = "Datatypes";
    fullName = "de/montiarcautomaton/lejos/lib/timer/
      Datatypes.cd";
  };

  @BumperBot!maa:MAAModelFile{
    simpleName = "BumperBot";
    fullName = "speedbumper/BumperBot.maa";
  };

  @Datatypes!cd(!4):CDModelFile{
    simpleName = "Datatypes";
    fullName = "de/montiarcautomaton/lejos/lib/motor/
      Datatypes.cd";
  };

  link unusedImports @BumperBot!maa(!2) -> @Datatypes!cd(!2);
  link unusedImports @BumperBot!maa(!2) -> @Datatypes!cd(!4);
  link unusedImports @BumperBot!maa -> @Datatypes!cd(!2);
  link unusedImports @BumperBot!maa -> @Datatypes!cd(!4);
}
```

Listing G.2: Ergebnis der Analyse zum Auffinden ungenutzter `imports`-Beziehungen.

Anhang H.

Lebenslauf

Name	Greifenberg
Vorname	Timo Herbert
Geburtstag	16.09.1985
Geburtsort	Aachen
Staatsangehörigkeit	deutsch
2013 - 2018	Wissenschaftlicher Mitarbeiter Lehrstuhl für Software Engineering RWTH Aachen University
2010 - 2013	Studium Master Informatik an der RWTH Aachen University
2006 - 2010	Studium Bachelor Informatik an der RWTH Aachen University
2005 - 2006	Zivildienst
2005	Städtisches Gymnasium Herzogenrath. Abschluss: Abitur

Abkürzungsverzeichnis

ACC	*Architecture Consistency Checking*
AD	*Activity Diagram*
AM	*Artefaktmodell*
API	*Application Programming Interface*
ARC	*Architecture Diagram*
ArtEx	*Artifact Data Explorer*
ArtViz	*Artifact Visualization*
AST	*Abstract Syntax Tree*
C&C	*Component-and-Connector*
CD	*Class Diagram*
CD4A	*Class Diagram for Analysis*
CI	*Continous Integration*
CLI	*Command Line Interface*
CoCo	*Context Condition*
CPS	*Cyber Physical System*
DEx	*Data Explorer*
DSL	*Domain Specific Language*
FSM	*Finite State Machine*
GPL	*General Purpose Language*
GUI	*Graphical User Interface*
IDE	*Integrated Development Environment*
JSON	*JavaScript Object Notation*
JVM	*Java Virtual Maschine*
M2M	*Model-to-Model*

M2T *Model-to-Text*

MA *MontiArc*

MAA *MontiArcAutomaton*

MDA *Model-Driven Architecture*

MDD *Model-Driven Development*

MDE *Model-Driven Engineering*

MontiArT *MontiCore Artifact Toolchain*

OCL *Object Constraint Language*

OCL/P *OCL Variante der UML/P*

OD *Object Diagram*

OM *Object Model*

OMG *Object Management Group*

PIM *Platform Independent Model*

PSM *Platform Specific Model*

RM *Reflexion Modeling*

RTE *Runtime Environment*

SC *Statechart*

UML *Unified Modeling Language*

UML/P *Unified Modeling Language usable for Programming*

VCS *Version Control System*

Abbildungsverzeichnis

2.1. Aufbau und Ablauf von MontiCore-basierten Generatoren. 12
2.4. Tabellarische Ansicht des DEx-Generators (entnommen aus [Rot17]). . . . 19

3.2. Darstellung von Abhängigkeiten mit Hilfe einer Abhängigkeitsmatrix (entnommen von [Str17b]). 33
3.3. Vertikale Tracebilityinformationen dargestellt als Graph unter Verwendung eines hierarchischen Layouts (entnommen von [Yak17b]). 34
3.4. Ausgewählte Elemente und ihre Nachbarschaft dargestellt als Graph unter Verwendung des Force-Directed-Layouts (erstellt mit [Mas17]). 35
3.5. Eine Softwarestadt aus dem Werkzeug *CodeCity* (entnommen aus [WL07]). 35
3.6. Die *Circle-based hierarchical view* von ARAMIS zur Darstellung von Architekturbeziehungen (entnommen aus [NLG+15]). 36
3.7. Die *Exploration View* von Sonargraph zur Darstellung von Architekturbeziehungen (entnommen von [Hel17b]). 37

4.1. Beispielszenario eines Softwareentwicklungsprojekts mit MAA (angepasst aus [BGRW18]). 42
4.2. Lösungsansatz zur artefaktbasierten Analyse modellgetriebener Softwareentwicklungsprojekte (angepasst aus [BGRW18]). 45

5.1. Übersicht über den Zusammenhang der verwendeten Sprachen und Modelle bei der Artefaktmodellierung. 54
5.2. Abstrakte Beziehungen zwischen Artefakten und die Organisation von Artefakten in Containern. 62
5.8. Ein System besteht aus Artefakten und ist entweder ein Werkzeug oder ein Produkt. 67
5.12. Module repräsentieren die architektonische Substruktur des Systems. Sie bestehen aus Artefakten und können sich aus Submodulen zusammensetzen. 69

6.1. Modellierung von Java-Artefakten und zugehörigen Beziehungen. 72
6.5. Java-Archive. 75
6.7. Java-Typen und deren Beziehungen. 76
6.11. Eine Sprache wird durch eine Grammatik und mehrere Wohlgeformtheitsregeln beschrieben. 81

6.16. CD-Modelldateien, verwandte Elemente und Beziehungen. 84
6.18. Generatoren sind Werkzeuge, die zu den Eingabesprachen passende Modelle einlesen und unter anderem aus FreeMarker-Templates und Java-Artefakten bestehen. 87
6.21. Aktionen, Events und deren Beobachtung. 90
6.24. Relevante Aktionen des Generierungsprozesses. 92
6.26. Eingabe- und Ausgabeartefakte von Werkzeugausführungen. 94
6.28. Template- und Java-Artefakte tragen zur Erstellung von Artefakten bei. . . . 96

7.1. Die Schritte der Gesamtmethodik zur artefaktbasierten Analyse. 111
7.2. Übersicht über die Teilschritte der artefaktbasierten Analyse. 111
7.13. Beispielhafte Durchführung des Mergealgorithmus unter Verwendung des `fullname`-Attributs als Zuordnungskriterium für Objekte. 126
7.16. Beispielhafte Anreicherung von Artefaktdaten. 129
7.17. Schematische Durchführung einer OCL-basierten Analyse am Beispiel der Change-Impact-Analyse für das Artefakt `@A5`. 131
7.18. Beispielhafte Filterung von Artefaktdaten. 133
7.19. Die Moduldefinition, zugeordnete Artefakte und deren Beziehungen, als Eingabe für die Berechnung der Ist-Architektur. 134
7.20. Oben links: Soll-Architektur; Unten links: Berechnete Ist-Architektur; Rechts: Berechnete Differenzarchitektur aus Soll- und Ist-Architektur. 135

8.1. Beispielhafte MontiArT-Instanz mit Zuordnung der Werkzeuge zu den Schritten der artefaktbasierten Analyse. 138
8.5. Schematische Darstellung der Klassenstruktur für ausführbare Jar-Archive. 143
8.10. Schematische Darstellung der Klassenstruktur der konkreten, statischen MontiArT-Extraktoren. 147
8.15. Beispielhafte Extraktion von Artefaktdaten mit dem `TemplateExtractor` aus einem Templateausschnitt. 151
8.23. Überblick über den Ablauf des `OCLGenerator`. 158
8.28. Überführung einer OCL-Prüfbedingung in eine Java-Prüfmethode (entnommen aus [Cuj17]). 161
8.30. Überführung einer OCL-Berechnungsvorschrift in eine Java-Methode zur Berechnung eines abgeleiteten Attributs (entnommen aus [Cuj17]). 162
8.35. Links: Die ArtViz Menüleiste; rechts: Der Zeitpunkt der Datenerstellung des ausgewählten Projekts. 169
8.36. Textuelle Darstellung von Artefaktdaten durch die Textansicht von ArtViz. . 170
8.37. Graph Visualisierung von Artefaktdaten mit ArtViz. 171
8.38. Oben: Details eines Knotens; unten: Details einer Kante. 171
8.40. Hierarchische Graphvisualisierung von Artefaktdaten mit ArtViz. 174
8.41. Farbcodierung zur Darstellung normierter Maßzahlen mit MontiArT. 175

8.42. Kuchendiagramm zur Darstellung von Metrikmaßzahlen mit ArtViz. 176
8.43. Balkendiagramm zur Darstellung von Metrikmaßzahlen mit ArtViz. 176

9.1. Methodik zur Verwendung der MontiArt-Werkzeugkette. 183
9.2. Methodik zur Extraktion von Beziehungen zwischen Artefakten mit Hilfe der Symboltabelle. 186
9.3. Das `Scope`-Interface und die `Scopes`-Klasse. 187
9.4. Entscheidungshilfe für die Verwendung und Anpassung der Reportinginfrastruktur. 189
9.13. Integration der Artefaktdatenanalyse in die bestehende Infrastruktur. 198

10.1. Die Soll-Architektur des DEx-Projekts. 200
10.8. Die Ist-Architektur des DEx-Projekts. 206
10.9. Die Differenzarchitektur des DEx-Projekts. 207
10.18. Überblick über den Ansatz zur automatisierten Instrumentierung von Makefiles und dem Beobachten Make-basierter Buildprozesse. 212
10.24. Artefaktbeziehungen eines beispielhaften, inkrementellen Generierungsprozesses mit MontiCore. 219

Listings

2.2. Ausschnitt aus dem Report `01_Summary`, erstellt für die MontiCore OD-Sprache. . . . 15
2.3. Beispielhaftes OD mit zwei hierarchisch verschachtelten Objekten. 17

5.3. AM: `Artifact`. . . . 63
5.4. AM: Fundamentale Beziehungen. . . . 63
5.5. AM: `ArtifactContainer`. . . . 65
5.6. AM: `Directory`. . . . 66
5.7. AM: `Archive`. . . . 67
5.9. AM: `System`. . . . 68
5.10. AM: `Product`. . . . 68
5.11. AM: `Tool`. . . . 68
5.13. AM: `Module`. . . . 69

6.2. AM: `JavaArtifact`. . . . 73
6.3. AM: `JavaSourceFile`. . . . 73
6.4. AM: `JavaClassFile`. . . . 74
6.6. AM: `JavaArchive`. . . . 75
6.8. AM: `Package`. . . . 76
6.9. AM: `Type`. . . . 78
6.10. AM: Beziehungen zwischen Java-Artefakten und -Typen. . . . 79
6.12. AM: `Language`. . . . 81
6.13. AM: `GrammarFile`. . . . 82
6.14. AM: `CoCo`. . . . 83
6.15. AM: `ModelFile`. . . . 83
6.17. AM: `CDModelFile`. . . . 85
6.19. AM: `FreeMarkerTemplate` und dessen Assoziationen. . . . 88
6.20. AM: `Generator`. . . . 89
6.22. AM: `Actions`. . . . 91
6.23. AM: `ExecutionProtocol`. . . . 91
6.25. AM: `Actions`. . . . 93
6.27. AM: Abgeleitete Ein- und Ausgaben bei der Werkzeugausführung. . . . 95
6.29. AM: Aktionen tragen zur Generierung von Artefakten bei. . . . 97

6.30. AM: Artefakte tragen zur Generierung anderer Artefakte bei. 98
6.31. Erweiterung des AMs zu Analysezwecken: Architekturrekonstruktion. . . . 100
6.32. Erweiterung des AMs zu Analysezwecken: Soll-Architektur. 100
6.33. Erweiterung des AMs zu Analysezwecken: Differenzarchitektur. 101
6.34. Erweiterung des AMs zu Analysezwecken: Indirekte Templatebeziehungen. 102
6.35. Erweiterung des AMs zu Analysezwecken: Ungenutzte Importe. 103
6.36. Erweiterung bestehender AM Klassen als Vorbereitung zur Analyse der neu auszuführenden Teile von Werkzeugketten. 104
6.37. Erweiterung des AMs zu Analysezwecken: Neu zu erstellende Artefakte. . . 105
6.38. Erweiterung des AMs zu Analysezwecken: Neu auszuführende Teile der Werkzeugkette. 105

7.3. OCL-basierte Spezifikation einer Abhängikeitsanalyse. 117
7.4. OCL-basierte Spezifikation einer Change-Impact-Analyse. 117
7.5. Beispielhafte Definition einer Soll-Architektur durch Spezifikation von Modulen und zugehörigen, erwarteten Beziehungen. 118
7.6. Beispielhafte Zuordnung von Artefakten zu Modulen (Moduldefinition). . . 119
7.7. OCL-basierte Spezifikation einer Metrik. 119
7.9. Definition der OCL-Bedingungen `deriveUnusedTool` zur Berechnung der ungenutzten Werkzeuge bei einer Werkzeugkettenausführung. 121
7.10. Beispielhaftes Tagdefinition zur Angabe der Nutzungsart von OCL Bedingungen. 121
7.11. Schema zur Berechnung von OD-Objektidentifikatoren. 122
7.12. Beispiel für Identifikatoren von Artefaktdaten. 123
7.14. Artefaktdaten als Eingabe zur schematischen Auswertung einer OCL-Prüfbedingung. 128
7.15. Gegenbeispiel als zusätzliches Ergebnis bei negativer Auswertung einer OCL-Prüfbedingung. 128
7.21. Ergebnis der beispielhaft durchgeführten Berechnung einer Metrik. 135

8.3. Aufruf von `ToolA` des ausführbaren Jar-Archivs `ExecutableJarArchive`. 142
8.6. `terminate`-Methode der `AToolCLI`-Klasse. 143
8.7. Shell-Skript zur Entfernung von Semikolons nach schließenden geschweiften Klammern bei Objektdefinition. 145
8.8. Schematischer Aufruf eines statischen Extraktors. 146
8.12. Schematischer Aufruf des `ArtifactContainerExtractors`. 149
8.17. Schematischer Aufruf des `ODMergers`. 153
8.19. Beispielhafte Spezifikation einer Konfigurationsdatei für den `ODMerger`. . 153
8.21. Beispielhafter Aufruf des `CDODConsistencyCheckers`. 157
8.25. Beispielhafter Aufruf des `OCLGenerators`. 159

8.26. Beispiele für OCL-Prüfbedingungen, die sich keinem einzelnen Kontext zuordnen lassen. 159
8.27. Beispielhafter Aufruf des `ODVerifiers`. 160
8.29. Beispielhafter Aufruf des `ODAccumulators`. 162
8.31. Beispielhafter Aufruf des `ODFilters`. 165
8.33. Beispielhafter Aufruf des `ArcReconstructors`. 167
8.34. Beispielhafter Aufruf des `ACCheckers`. 168
8.44. Schematischer Aufruf des `ODFlatteners`. 177
8.46. Beispielhafter Aufruf des Werkzeugs `OD2JSON`. 179
8.47. Beispielhafter Aufruf des `CDOCLSplitters`. 179
8.48. Implementierung des `CDOCLSplitters`. 180

9.5. Abbruchfunktion bei Nutzung einer Shell-Skript-basierten MontiArT-Instanz. 190
9.6. Beispielhafter Aufruf der Abbruchfunktion des mitgelieferten Shell-Skripts. 190
9.7. Erweiterte Fehlerbehandlung, bei der die Ausführung der Werkzeugkette im *failure*-Fall unter Ausgabe einer Warnung fortgesetzt wird. 191
9.8. Fehlerbehandlung mit Make. 192
9.9. Konfiguration des initial ausgewählten Projekts mit Hilfe der `global-Properties.json`-Datei. 192
9.10. Projektspezifische Einstellungen zur Konfiguration der Menüleiste, der textuellen Ansicht und der Graphansicht mittels `properties.json` 193
9.11. Konfiguration von Metriken per `properties.json` Datei. 194
9.12. Die Ordnerstruktur des `assets`-Ordners von ArtViz. 195

10.2. Exemplarischer Aufruf der Methode `replaceTemplate` zum Ersetzen von Templates in DEx. 201
10.4. Ergebnis der Extraktion für den exemplarischer Methodenaufruf aus Listing 10.2. 202
10.5. Änderung der Berechnungsvorschrift für die `contibutesTo` Beziehung für `JavaMethodCalls`. 203
10.13. Die Erweiterung des AMs für die Eingabeartefakte des BumperBot-Projekts. 210
10.14. Analyse von ungenutzten Beziehungen für `MAAModelFiles`. 211
10.15. Ungenutzte `imports`-Links der BumperBot-Eingabeartefakte. 211
10.16. Analyse von ungenutzten Artefakten für die BumperBot-Applikation. . . . 211
10.19. Erweiterung des Artefaktmodells für das Monitoring Make-basierter Buildprozesse. 213
10.20. Beispiel für eine Make-Regel. 214
10.21. Angepasste Make-Regel nach der Instrumentierung. 214
10.22. Werkzeug zur automatisierten Instrumentierung von Makefiles. 215
10.23. Aufruf von eines Makefile mit beliebigem Namen. 216
10.25. Initiales Makefile für die inkrementelle Generierung mit MontiCore. 220

10.26. Anpassungen Gegenüber dem Makefile aus Listing 10.25 für eine korrekte inkrementelle Generierung mit MontiCore. 221
10.27. Anpassung der Regel `automaton-tool.jar`, um auf das Entfernen von Eingabedateien reagieren zu können. 222
10.28. Anpassungen des Makefile für die effiziente inkrementelle Generierung mit MontiCore. 224
10.29. Ausschnitt eines generierten Shell-Skripts zur optimierten Prüfung auf relevante Änderungen von handgeschriebenem Quellcode. 224
10.31. Projektspezifische Erweiterung des Artefaktmodells bei der Testfallgenerierung im Automotivekontext. 228
10.32. Die OCL-basierte Analyse ermöglicht das Tracing von Anforderungen zu Testfällen. 229
10.33. Die OCL-basierte Analyse ermöglicht die Rekonstruktion der Funktionsarchitektur. 229

C.1. AM für MontiCore-basierte MDD-Projekte. 261

D.1. Tagschema für artefaktbasierte Analysen. 275

F.1. Shell-Skript zur Definition einer ausführbarenMontiArT-Instanz. 279
F.2. Kombination aus Moduldefinition und der Soll-Architektur für das DEx-Projekts. 285
F.3. Berechnete Ist-Architektur des DEx-Projekts. 297
F.4. Berechnete Differenzarchitektur des DEx-Projekts. 300

G.1. Extrahierte Artefaktdaten für die Eingabeartefakte beider Varianten des BumperBots. 303
G.2. Ergebnis der Analyse zum Auffinden ungenutzter `imports`-Beziehungen. 308

Tabellenverzeichnis

3.1. Beispiele bekannter Metriken aus der Klasse der Produkt- und Testüberdeckungsmetriken. 31

7.8. Tags zur Angabe der Nutzungsart von OCL-Bedingungen. 120

8.2. Mögliche Ein- und Ausgaben von MontiArT-Werkzeugen. 141
8.4. Standardoptionen von MontiArT Werkzeugen. 142
8.9. Mögliche Option beim Aufruf statischer Extraktoren. 147
8.11. Durch den `ArtifactContainerExtractor` extrahierte AM-Anteile. . 148
8.13. Mögliche Optionen bei Aufruf des `ArtifactContainerExtractors`. 149
8.14. Durch den `TemplateExtractor` extrahierte AM-Anteile. 150
8.16. Durch den `JavaExtractor` extrahierte AM-Anteile. 152
8.18. Mögliche Optionen bei Aufruf des `ODMergers`. 153
8.20. Fehlerarten bei der Konsistenzprüfung und deren Ursachen. 156
8.22. Mögliche Option bei Aufruf des `CDODConsistencyCheckers`. 157
8.24. Geplante und bereits prototypisch generierbare Werkzeuge der `ocl-tools`. 158
8.32. Mögliche Optionen bei Aufruf des `ODFilters`. 165
8.39. Liste der mittels *Optionsleiste* wählbaren Einstellungen der *Graphansicht* von ArtViz. 172
8.45. Mögliche Option bei dem Aufruf des `ODFlatteners`. 177

10.3. Durch den `JavaExtractor` extrahierte AM-Anteile. 201
10.6. Durch den `DExActionReporter` extrahierte AM-Anteile. 204
10.7. Moduldefinition des DEx-Projekts: Zuordnung von Ordnern zu Modulen. . 205
10.10. Untersuchte Artefakte des DEx-Projekts 208
10.11. Untersuchte Beziehungen zwischen Artefakten des DEx-Projekts. 208
10.12. Beobachtete Aktionen während der Ausführung des DEx-Generators. . . . 209
10.17. Die Ergebnisse der BumperBot-Analysen in Zahlen. 212
10.30. Ausführungsdauern (Minimum, Maximum, Durchschnitt) in ms für den Aufruf der `gen1`-Regel der verschiedenen Makefile-Varianten. 226

A.1. Die in Abbildungen und Listings verwendeten Markierungen. Markierungen geben die verwendete Sprache an. 257

A.2. Die in Abbildungen und Listings verwendeten Stereotypen. Stereotypen sind Markierungen zugeordnet. Sie geben den Verwendungszweck an. 258

B.1. Zuordnung von Anforderungen an Analyseverfahren zu den Abschnitten, welche die Anforderungserfüllung beschreiben. 259
B.2. Zuordnung von Anforderungen an die Modellierungstechnik zu den Abschnitten, welche die Anforderungserfüllung beschreiben. 260
B.3. Zuordnung von Anforderungen an die Werkzeugunterstützung zu den Abschnitten, welche die Anforderungserfüllung beschreiben. 260

E.1. Verteilung der mitgelieferten MontiArT-Werkzeuge auf die ausführbaren Archive. 277

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06, GKR+08] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve reliability of adaprivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the two books [Rum16] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency

of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called

sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct

variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also

for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

]Literaturverzeichnis

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the*

ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13), pages 444–454. ACM New York, 2013.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.