Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Ausarbeitung Anwendungen 2 - SoSe 2009

Thorben Pergande

Automatisierte Architekturanalyse mittels UML 2.0 Diagrammen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

1 Einführung in das Themengebiet

In dieser Ausarbeitung wird der aktuelle Stand der Forschung und vergleichbare Arbeiten zu dem Thema der automatisierten Architekturanalyse mittels UML2.0 Diagrammen beschrieben. Im Folgenden wird eine Einführung in das Themengebiet der angestrebten Architekturanalyse und die damit zusammenhängenden Fragestellungen dargestellt. Kapitel 2 befasst sich mit vergleichbaren Arbeiten, die Bezug zu den offenen Fragestellungen und zu dem Thema allgemein haben. Abgrenzungen zu den eigenen Arbeitszielen werden in Kapitel 3 dargestellt. Kapitel 4 fasst die Ergebnisse dieser Ausarbeitung zusammen und zeigt die nächsten Schritte auf.

1.1 Einführung in das Themengebiet

Die Architektur eines Softwaresystems stellt die Abstraktion der konkreten Realisierung der Anforderungen dar. Dabei beschreibt die Architektur als grobes Systemkonzept den Schnitt der benötigten Komponenten und Schnittstellen, als auch die Beziehungen dieser zueinander. Der Quelltext eines Systems stellt dabei den zum jeweiligen Zeitpunkt aktuellen Stand der Architektur, die sog. IST-Architektur dar. Während der frühen Softwareerstellungsphasen, z.B. während der Anforderungs- oder Designanalyse, kann eine erste Version der geplanten Architektur mittels verschiedener Modellierungssprachen erstellt werden. Diese Architektur kann sich im Laufe des Softwareerstellungsprozesses erweitert oder geändert werden. Diese Darstellung der Architektur mittels eines oder mehreren Diagrammen beschreibt die SOLL-Architektur. Eine mögliche und für diese Ausarbeitung genutzte Modellierungssprache ist UML 2.0 (Kecher (2005)), welche von der OMG (Group (2009)) definiert und bei vielen Anwendern und Entwicklern bekannt ist.

Der Modellierungsstandard UML umfasst verschiedene Möglichkeiten, die Architektur eines System darzustellen, die sog. Sichten. Man kann zwischen 4 verschiedenen Sichten auf die Architektur unterscheiden, diese werden im Folgenden mit den dazugehörigen Diagrammtypen dargestellt:

- statische Sicht: beschreibt die Komponenten des Systems und Beziehungen zueinander
  mögliche UML Diagramme: Klassen-, Komponenten- und Paketdiagramm
- Fachliche Sicht: beschreibt die Anforderungen und Prozesse eines Systems
  mögliche UML Diagramme: Sequenz-, Aktivitäts- und Anwendungsfalldiagramme
- Verteilungssicht: beschreibt die Verteilung der statischen Sicht auf verschiedene Computer, Prozesse oder Netzwerke
  mögliche UML Diagramme: Verteilungsdiagramm

- Laufzeitsicht: beschreibt die Abläufe der Synchronisation oder Datenaustausch innerhalb der Architektur
  mögliche UML Diagramme: Sequenz- und Interaktionsdiagramm

Da die SOLL-Architektur eine Art Bauanleitung für das System und die IST-Architektur (Quelltext) die jeweils aktuelle Realisierung der Architektur darstellt, muss stetig geprüft werden, ob die beiden Formen der Architektur miteinander übereinstimmen. Dieser Vergleich wird als Architekturanalyse bezeichnet. Differenzen zwischen der SOLL- und IST-Architektur entstehen dadurch, dass die IST-Architektur (Quelltext) manuell bearbeitet wird und während der Bearbeitung Abweichungen zur geplanten SOLL-Architektur entstehen. Dies kann durch Zeitdruck im Projekt, mangelnde Kommunikation der SOLL-Architektur oder aufgrund mangelnden Verständnisses dieser folgen.

Derzeit verbreitet sind metrikenbasierte Ansätze zur Überprüfung der Architektur. Dabei ermittelt ein Werkzeug automatisiert aus dem Quelltext verschiedene Kennzahlen. Anhand dieser Kennzahlen kann ein Analyst manuell einen Vergleich mit der Soll-Architektur vornehmen. Das Ergebnis einer metrikenbasierten Analyse ist zum größten Teil eine Sammlung von Metriken, also zu interpretierende Zahlen. Diese Zahlen sprechen nicht für sich selbst, sodass nicht davon ausgegangen werden kann, dass der Auftraggeber die Ergebnisse selbstständig analysieren und auswerten kann.

Die Idee für eine automatisierte Architekturanalyse mittels UML 2.0 Diagrammen soll genau diese selbstständige Analyse ermöglichen. Dabei ist die Vision, dass die Eingaben für die Analyse der Quelltext und ein oder mehrere Diagramme, die die SOLL-Architektur beschreiben, sind.

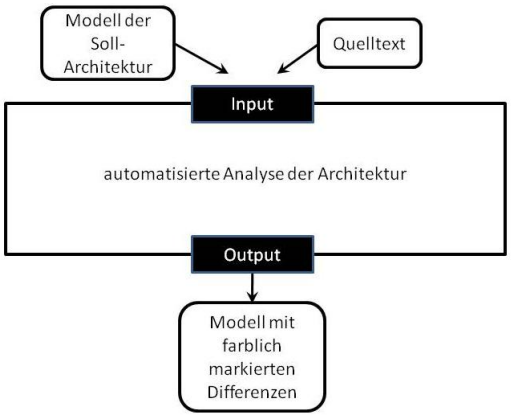

Abbildung 1: Black-Box der angestrebten Architekturanalyse

Das Ergebnis der Analyse soll ein UML-Diagramm vom selben Typ wie beim Input sein, das Differenzen zwischen der SOLL- und IST-Architektur farblich hervorhebt. Der Vorteil dieses

Ansatzes ist der Umgang mit dem Ergebnis, da UML Diagramme auf Auftraggeberseite als bekannt vorausgesetzt werden. Eine Interpretation der Ergebnisse kann nun schneller und mit den bereits eingesetzten Modellieungswerkzeugen erfolgen.

Um diesen Ansatz der Architekturanalyse zu verfolgen, muss eine Möglichkeit gefunden werden, wie Quelltext mit UML 2.0 Diagrammen verglichen werden und wie das Ergebnisdiagramm erstellt werden kann. Für den Vergleich zwischen Quelltext und UML 2.0 Diagrammen werden zwei verschiedenen Ansätze verfolgt, die den Ablauf in Abbildung 1 näher definieren.

Bottom-Up Ansatz

Bei diesem Ansatz soll aus dem Quelltext ein UML 2.0 Diagramm hergeleitet werden, das denselben Typ hat, wie das Input Diagramm. Anschließend sollen das Ergebnis aus einem Vergleich zwischen dem hergeleiteten und dem eingegebenen Diagramm erfolgen. Als Ausgabe wird das hergeleitete Diagramm mit farbigen Akzenten bei Differenzen verwendet.

Top -Down Ansatz

Hier soll aus der SOLL-Architektur Quelltext generiert werden, der dann mit der IST-Architektur verglichen werden soll. Dazu kann auf Mechanismen der Code Generierung zurückgegriffen werden (Quellen). Differenzen in zwischen SOLL- und IST-Architektur werden in eine Kopie des Input-Diagramms eingezeichnet und als Ergebnis der Analyse veröffentlicht.

Probleme und Fragestellungen

Die modellgetriebene Architektur (Stahl u. a. (2007)) bietet theoretische Varianten, die für die Umsetzung genutzt werden können, und unterscheidet zwischen Modell-Stufen. Um vergleichen zu können, müssen die Komponenten sich auf derselben Stufe befinden. Beide Varianten setzen eine Model-Transformation voraus. Für Bottom-Up werden Methoden zur Transformation von Quelltext zu Diagramm benötigt, entsprechend umgekehrt für Top-Down. Für die konkrete Umsetzung waren umfangreiche Recherchen notwendig, um existierende Ansätze und Lösungen für die Einzelschritte zu belegen.

2 Vorstellung vergleichbarer Arbeiten

In diesem Kapitel werde Ausarbeitungen vorgestellt, die in einem ähnlichen Kontext des hier vorgestellten Themas sind.

2.1 UML-based Design Test Generation

Die Autoren Waldemar Pires, Joao Brunet und Franklin Ramalho, allesamt von der Federal Univerity of Campina Grande in Brazilien, beschreiben einen Ansatz zur automatisierten Konformitätsprüfung von UML Klassendiagrammen und Java-Implementationen (Pires u. a. (2008)). Dabei werden Werkzeuge genutzt, um Design-Regeln als JUnit-Tests zu generieren. Der bereits in vorherigen Veröffentlichungen vorgestellte "Design Wizard" bietet Methoden zur Sammlung von Struktur-Informationen des Quelltexts, zum Beispiel über verwendete Klassen, Methoden und deren Referenzen zueinander. Aufgrund dieser Basis-Informationen können manuell formulierte Unit-Tests über JUnit durchgeführt werden, die Aussagen über Einhaltung und Nicht-Einhaltung von Design-Regeln ermöglichen.

Code 1: DesignTest

```
1 public void testCommunication() {
2   DesignWizard dw;
3   dw = new DesignWizard(``project.jar'');
4   designwizard.ui.Class clazz;
5   clazz = dw.getClass(``A'');
6   Set<String> usedBy;
7   usedBy = clazz.getClassesUsedBy();
8   assertFalse(usedBy.contains(``B''))
9 }
```

Abbildung 2: Beispiel für einen Design-Test

Um im nächsten Schritt die Design-Regeln automatisiert aus konkreten Klassen-Diagrammen zu erstellen, verwenden die Autoren Methoden der modellgetriebenen Softwarearchitektur - MDA. Dazu wird das ATL-Framework (Nantes (2008)) genutzt, basierend auf Modellen, Meta-Modellen und Transformationen, um Klassendiagramme im XMI-Format mit zuvor definierten Regeln in JUnit-Tests zu transformieren. Das Klassendiagramm, also das Modell, muss dazu einem zuvor hinterlegten UML2-Metamodell genügen. Entwickler oder Architekten können damit ad-hoc Tests auf den gerade erstellten Quelltext anwenden. Testergebnisse werden dabei farblich dargestellt, grün bei positiven, rot bei negativen. Zukünftig sollen auch weitere UML2-Diagramme, zum Beispiel Aktivität und Interaktion, verarbeitet werden können. Zudem sollen OCL-Ausdrücke in die Design-Tests aufgenommen werden.

2.2 On reverse engineering an object-oriented code into UML class diagramms incorporation extensible mechanisms

Für die Rekonstruktion und Analyse eines Produktes ist es interessant herauszufinden, aus welchen statischen Aspekten eine Realisierung aufgebaut ist. Für diese Fragestellung haben

die Autoren Vinita, A. Jain und D. Tayal eine Lösung beschrieben, wie aus einem Objekt-orientierten - Quelltext ein UML Klassen-Diagramm hergeleitet werden kann (Vinita u. a. (2008)). Dieses Ziel wurde bereits in anderen Ausarbeitungen, z.B. (Sutton und Maletic (2005)), beschrieben, Vinita u.a. erweitern diese Ziele so, dass versucht werden soll, alle möglichen Elemente der Klassendiagramme einzubinden, die bisher noch nicht betrachtet wurden. Beispielsweise sind damit "tagged-value" gemeint. Hierzu stellen die Autoren einen Algorithmus vor, der diese Umwandlung vornimmt. Der Algorithmus benötigt als Eingabe den Quelltext und gibt als Ausgabe das dazugehörige UML-Klassendiagramm aus. Inhaltlich ist der Algorithmus in zehn Schritte mit insgesamt 15 Regeln unterteilt.

Aus Platzgründen werden nun einige für den Analysekontext relevanten Regeln vorgestellt.

- Regel 1: Finde klassenbeschreibende Schlagwörter wie "class" im Quelltext und erstelle dazu eine UML Klassen mit dem Namen, Attributen und Methoden.
- Regel 2: Finde Schlagwörter, die eine abstrakte Klasse beschreiben und erweitere die Klasse um das Attribut "abstract".
- Regel 6: Finde gruppierende Elemente mit den Schlagwörtern "package" oder Header-Dateien und zeichne diese entsprechend in das Diagramm ein.
- Regel 10: Finde Assoziationen, dabei wird geprüft, welche Klasse welche anderen Klassen nutzt (Assoziation), verwenden sich zwei Klassen gegenseitig wird dies als bidirektionale Beziehung eingezeichnet.
- Regel 11, 12: Finde Aggregationen und Kompositionen, ähnlich wie Regel 10.
- Regel 13: Finde Vererbung, z.B. über das Schlagwort "implements" oder über ":".
- Regel 15: Identifiziere Erweiterungsmechanismen: Hier sollen Klassendiagrammerweiterungen wie "tagged values" oder OCL-Constraints erkannt werden. Dazu muss für jede eingesetzte Sprache ein Konstrukt identifiziert werden, die solche Constraints darstellen.

Für jede zu unterstützende Sprache müssen die Regeln angepasst werden, da die zu identifizierenden Schlagwörter unterschiedlich sind, z.B. "package" in Java und "namespace" in .Net zur Identifizierung gruppierender Elemente. Dieser Ansatz wurde in der Ausarbeitung von Vinita et al. geprüft, indem ein Beispielquelltext in ein Klassendiagramm überführt wurde. Hierzu befindet sich ein Auszug aus der originalen Ausarbeitung in Abbildung 3.

Aus Sicht der Autoren kann mit solch einem Algorithmus zunächst aus einer konkreten Implementation ein Modell hergestellt werden, mit diesem Modell kann anschließend mit Hilfe von Code-Generierung eine andere Implementierungssprache gewählt werden.

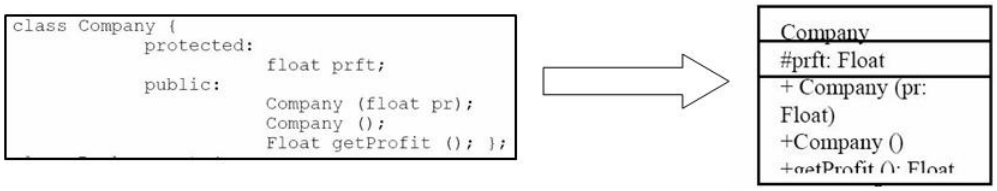

Abbildung 3: Beispiel für eine Überfühgun von Quelltext zum Modell

2.3 Metamodel Approach on Model Conformance and Multiview Consistency Checking

Verschiedene Modellierungssprachen wir BON (Waldén und Nerson (1995)) oder UML werden in der MDA für die Spezifikation der Softwaresysteme genutzt. Die Autoren Chen Shu, Qu Guo Qing und Xioa Jing befassen sich in (Shu u. a. (2008)) damit, wie die Konsistenz von Modellen formal beschrieben und geprüft werden kann. Ebenfalls untersuchen die Autoren die Möglichkeit, wie die dynamischen und statischen Sichten auf ein Szenario so verglichen werden können, dass diese miteinander kompatibel sind. Aus der MDA ist der Einsatz von Meta-Modellen gebräuchlich, die die Struktur eines Modells, z.B. Komponentendiagramms, beschreiben. Als zu untersuchende Modellierungssprache wird BON genutzt, BON unterscheidet zwischen statischen Abstraktionen, z.B. Klassen, und statischen Beziehungen, z.B. Assoziationen. Die statischen Abstraktionen werden mittels logischer Constraints als Gruppen definiert.

Abbildung 4 zeigt beispielhaft solch einen logischen Constraint, der sicherstellt, dass Namen für Klassen nur einmal für das gesamte System gewählt werden können.

$$\textbf{Unique\_Feature\_Name\_Assertion} \mathrel{\widehat{=}}$$
$$(\forall c \in CLASS)((f_1 \in Class\_Features\ (c)$$
$$\wedge f_2 \in Class\_Features\ (c) \Rightarrow$$
$$Feature\_Name(f_1) = Feature\_Name(f_2)) \Rightarrow (f_1 = f_2))$$

Abbildung 4: Beispiel eines logischen Contraint des abstrakten statischen Automaten

In dieser Form wird ein gesamtes Meta-Modell für einen Modellierungsaspekt erstellt. Das Ergebnis ist ein abstrakter Automat, mit Hilfe dessen Modelle aus dieser Kategorie gegen das erstellte Meta-Modell geprüft werden, ob die definierten Regeln eingehalten und somit das Model mit dem Model konsistent ist. Für statische Beziehungen wird äquivalent ein Automat erstellt. Ein weiterer Aspekt dieser Arbeit liegt darin, die dynamischen Sichten, bestehend aus Objekte, Nachrichten und Abläufen, mit der statischen Sicht zu vergleichen. Dabei wird durch die Autoren besonders Wert auf die Konsistenz zwischen Klassen/Objekten und Nach-

richten/Routinen gelegt. Dazu werden die bestehenden abstrakten Automaten der statischen Sicht um Constraints erweitert, die solche Prüfungen durchführt. Abbildung 5 zeigt eine solche Erweiterung.

$$\textbf{Class\_and\_Object\_Assertion} \mathrel{\widehat{=}} (\quad o \in \mathrm{OBJECT})(o \in \mathrm{union}(\mathrm{CLASS})).$$

Abbildung 5: Erweiterung um Multiview

Dieser Constraint stellt sicher, dass ein Objekt aus einer dynamischen Sicht auch in der statischen Sicht enthalten ist. Sollte dies nicht der Fall sein, ist entweder ein Objekt hinzugefügt oder nicht implementiert worden. Beide Fälle würde die Konsistenz zwischen den Sichten verletzten. Die Ausarbeitung zeigt viele weitere Constraints für die abstrakten Automaten auf. Derzeit haben die Autoren nicht alle Aspekte, die mitteln BON modelliert werden können, betrachtet. In zukünftigen Arbeiten sollen die fehlenden Aspekte realisiert werden.

2.4 Automatic Code Generation: A Practical Approach

George A. Papadopoulos zeigt in seiner Ausarbeitung (Papadopoulos (2008)), wie Techniken aus der MDA eingesetzt werden können, um aus UML-Modellen Quelltext für die Sprache "Manifold" automatisiert zu erstellen. Dabei stellt er verschiedene Werkzeuge vor, die für dieses Vorhaben genutzt werden können.

Um ein Diagramm zu erstellen, sollten lt. G. Papadopoulos zunächst die Objekte der höchsten Ebene identifiziert und modelliert werden. Für die Code-Generierung wird in diesem Schritt ebenfalls die Startmethode für die Realisierung am Modell definiert. Danach sollen für jede Komponenten die Operationen und Variablen gefunden und integriert werden. Weitere Design-Vorgaben werden in der Ausarbeitung dargestellt. Um nun von Modellen zu Quelltexten zu kommen, werde wie bereits in Arbeit (Pires u. a. (2008)) dargestellt, sog. Modell-Transformationen benötigt. Dies sind Regeln, wie ein Diagramm als Quelltext dargestellt werden soll. Solche Regeln müssen pro Diagrammtyp und Programmiersprache definiert werden, sodass für ein breites Spektrum an nutzbaren Modellen/Sprachen ein großer Aufwand entsteht. Aus der Ausarbeitung geht hervor, dass solche Regeln nicht automatisiert, sondern manuell erstellt werden müssen. Abbildung 6 zeigt als Übersicht, wie solch eine Transformation in diesem Beispiel vorgenommen werden kann. Des Weiteren stellt G. Papadopoulos vor, wie auch Verhaltensdiagramme, z.B. Sequenzdiagramme zur Code-Generierung eingesetzt werden können. Dazu stellt der Autor wie bei den statischen Modellen Design-Vorgaben für die Modellierung vor, bei deren Einhaltung ebenfalls Transformationsregeln für die Konversion erstellt werden können.

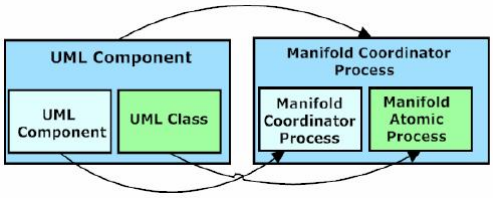

Abbildung 6: Mappingmodell für Komponenten auf Manifold

Als Eingabe für diesen Beweis der Machbarkeit werden die Modelle als XMI-Format genutzt. Diese werden dann zunächst in ein Manifold Metamodel und daraus dann in Manifold Quelltext überführt. Durch den Einsatz von XMI als Eingabeformat ist die Erstellung der Modelle fast Werkzeugunabhängig, solange das jeweilige Werkzeug einen XMI-Export anbietet. XMI basiert auf einer XML-Struktur und kann somit mit bekannten XML-Werkzeugen und -Methoden, wie XPath (W3C (1999)) etc. bearbeitet werden.

Weiterführend sollen die enthaltenden Methoden so erweitert werden, dass auch eine Konsistenzprüfung und eine Validierung des eingegebenen Modells vorgenommen wird. In dieser Ausarbeitung wird das eingegebene Modell nicht auf Korrektheit oder Schwächen überprüft, sonder als korrekt und Basis der weiteren Schritte behandelt.

3 Abgrenzung zu eigenen Arbeitszielen und / oder Methoden

In (Pires u. a. (2008)) und (Papadopoulos (2008)) werden als Import die Modelle im XMI-Format genutzt. XMI basiert auf XML-Strukturen und kann mit entsprechenden Methoden verarbeitet werden. Dies ist für die Architekturanalyse eine Vereinfachung, da solch ein Format eine Lösung für die automatisierte Bearbeitung von Modellen darstellt. Farbliche Akzente, Vergleiche und auch die automatisierte Erstellung können über Manipulation der XMI-Daten geschehen und es ist nicht mehr nötig, Modellierungswerkzeuge aus dem Programmablauf heraus fernzusteuern wie es z.B. Interop Assemblies bei MS Visio (Microsoft (2008)) ermöglichen.

Für die UML2.0-basierte Architekturanalyse soll somit als Input und Output das Modell per XMI vorliegen. Der Einsatz dieses Formats setzt voraus, dass die vom Designer und Architekten eingesetzten UML Werkzeuge einen Import/Export dafür bereitstellen.

Die in Ausarbeitung von Pires et al. vorgestellte Methode stellt eine automatisierte modellbasierte Architekturanalyse dar. Derzeit umfasst die dargestellte Arbeit eine Lösung für Klassendiagramme und Java-Implementationen. Weitere Modelle zu unterstützen ist in nachfolgenden Arbeiten geplant. Die Methodik ähnelt dem Top-Down-Ansatz des in Kapitel 1 dargestellten

Ansatzes. Pireset al. zeigen, wie man die Transformation vom Modell hin zu einem speziellen Quelltext realisieren kann.

Hierbei handelt es sich im Gegensatz zu den Zielen der UML2.0-basierten Architekturanalyse um Unit-Tests, die auf die IST-Architektur angewandt werden. Dieses Vorgehen bietet eine weitere Möglichkeit, wie man SOLL- mit IST-Architektur vergleichen kann, entspricht aber nicht den Zielen der uml-basierten Architekturanalyse. Der Unterschied ist deutlich bei der Art des Vergleichens der Architektur und der Repräsentation der Ergebnisse sichtbar. Bei Pires et al. übernimmt das Junit-Framework die Visualisierung der Ergebnisse, grün bei Erfolg und rot sobald ein Fehlerfall auftritt. Diese Ergebnisse sind eher an Entwickler adressiert, ebenso die Erstellung der Design-Tests. Der in Kapitel 1 vorstellte Ansatz soll ein Kommunikationsmedium über die verschiedenen Rollen eine Softwareerstellungsprojekts hinweg darstellen und nutzt deshalb als Ergebnis akzentuierte UML-Diagramme. Die Anpassung der Daten, sodass die Akzente gesetzt werden, ist im Einzelnen noch zu realisieren.

(Vinita u. a. (2008)) stellen einen Algorithmus vor, der genau eine Lösung auf die Frage: " Wie generiert man aus Quelltext ein Diagramm?" für Klassendiagramme beantwortet. Die Struktur des Algorithmus ist so gestellt, dass eine Erweiterung und Anpassung für weitere Modelltypen, z.B. Komponentendiagramme denkbar sind. Komponentendiagramme stellen eine praktikable Modellierungsdarstellung für Architekturen dar, sodass der Schnitt der Komponenten und deren Beziehungen zueinander vergleichsweise einfach zu realisieren sind. In den nächsten Schritten dieser Ausarbeitung soll validiert werden, ob eine Änderung des Algorithmus hin zu Komponentendiagrammen möglich ist. Dies würde für zwei mögliche UML2.0 Diagramme den Bottom-UP-Ansatz der UML-basierten Architekturanalyse (vgl. Kapitel 1) ermöglichen. Dieser Algorithmus stellt eine Transformationsregel für einen gezielten Diagrammtyp dar, solche Transformationsregeln zwischen Modellen eines Softwaresystems finden sich auch in dem Ansatz MDA. Auch wenn der Ansatz der automatisierten Architekturanalyse nicht unbedingt auf den MDA-Ansatz versteift sein soll, ist aufgrund der gemeinsamen Probleme, die automatisierte Transformation von Modellen, für genau diese Problemfelder die Mechanismen der MDA einsetzbar.

Diese Mechanismen werden auch in den Ausarbeitungen (Pires u. a. (2008)) und (Papadopoulos (2008)) genutzt. Alle drei Ausarbeitungen sind aus dem Jahr 2008 und zeigen, wie der MDA-Ansatz für die Softwareentwicklung/Softwarearchitekturanalyse aktuell realisiert werden kann. In allen Ausarbeitungen wird deutlich, dass die Transformation zwischen Modellen (z.B. Quelltext und UML-Diagramm) manuell realisiert werden müssen.

Noch offen sind die Fragen, wie Modelle miteinander verglichen werden, um Differenzen zu erkennen. Chen Shu (Shu u. a. (2008)) gibt für einen spezialisierten Kontext eine Möglichkeit dafür an. Die Prämissen von ChenShu, dass alle Modelle formal geprüft werden müssen, gelten nicht für diese Ausarbeitung, da der Formalismus vor allem darauf zielt, dass die erstellten Diagramme dem jeweiligen Standard genügen. Bisher wird im Ansatz aus Kapitel 1 davon

ausgegangen, dass das eingehende Modell korrekt ist, und als Ziel ist herauszufinden, ob die Realisierung (IST-Architektur) dem Model genügt. Ein formales Vorgehen gibt dem Ersteller einen engen Rahmen, der Ersteller könnte bewusst gegen formale Anforderungen verstoßen um die Aussagekraft eines Diagramms zu erhöhen. Bis zu einem gewissen Grad soll für die uml-basierte Architekturanalyse diese Freiheit gewahrt bleiben. Vorbedingungen sind jedoch, dass nur Elemente in einem Modell aus einem Meta-Modell genutzt und nicht Elemente verschiedener Diagrammtypen (z.B. Klassen- und Aktivitätsdiagramm) gemischt werden. Dies kann rein technisch wegen der manuell zu erstellenden Transformationsregeln nicht gewährleistet werden. Des Weiteren nutzt ChenShu keine automatisierten Techniken zur Validierung, solche Methoden werden benötigt, Ergebnisse der Analyse generiert werden können.

4 Zusammenfassung

Der Schwerpunkt dieser Arbeit ist einen Überblick über andere Arbeiten zu geben, die eine Schnittmenge mit dem in Kapitel 1 gezeigten Ansatz haben.

Vor allem die Fragen, wie Diagramme in Quelltexte und in die entsprechend andere Richtung transformiert werden können und wie die Ergebnisse der Transformation verglichen werden kann, stellen das aktuelle Kernproblem der uml-basierten Architekturanalyse dar. Als Ergebnis der Recherche ist zu bemerken, die Transformation der Modelle hauptsächlich im Kontext des MDA-Ansatzes zu finden ist. Solche Transformationen, bzw. die Regeln der Abbildung, müssen manuell erstellt werden. Das Klassendiagramm wird hierbei sehr oft als Beispieldiagramm genutzt, wobei für eine Architekturanalyse damit nur ein mögliches Diagramm der statischen Sicht berücksichtigt wird. Gerade Verhaltensdiagramme, z.B. Aktivitätsdiagramme, die die wichtige fachliche Sicht einer Architektur darstellen, werden jedoch wenig beachtet. Hierbei ist das Abbilden der Modelle auf Quelltext noch nicht geklärt.

XMI bietet als Import- und Exportformat vieler Modellierungswerkzeuge eine plattform- und werkzeugunabhängige Möglichkeit, wie Diagramme automatisiert verarbeitet und angepasst werden können. XMI basiert auf XML und kann somit auch mit den entsprechenden Werkzeugen verarbeitet werden, sodass z.B. farbige Akzente setzen in den Programmablauf integriert werden kann. Des Weiteren kann das Ergebnis der Architekturanalyse mittels UML2.0-Diagrammen dann in das beim Auftraggeber eingesetzte Modellierungswerkzeug importiert werden, vorausgesetzt XMI wird unterstützt.

Auf die Fragestellung, wie man nach den nötigen Transformationen Modelle miteinander vergleichen kann, wurde in der Recherche keine Antwort gefunden. Eine Ausarbeitung zu diesem Thema hat als Schwerpunkt die Einhaltung der jeweiligen Diagrammspezifikationen mit Hilfe von formalen Regeln, was aber für die Architekturanalyse nicht im Mittelpunkt steht. Ausarbeitung verzichtet auf den detaillierten Bezug auf Werkzeuge, und befasst sich mit Ansätzen und

Möglichkeiten für eine automatisierte Architekturanalyse. Einzelne Werkzeuge werden in den verschiedenen Ausarbeitungen aus Kapitel 1 vorgestellt und können in nächsten Schritten betrachtet werden. Eine kurze Recherche zu Werkzeugen hat ergeben, dass für Klassendiagramme schon bestehende Lösungen für das Forward und Reverse- Engineering gibt. Die nächsten Schritte werden die Realisierung einer Modelltransformation für Komponentendiagramme und die Suche nach einer Lösung für den Vergleich von Modellen sein.

Literatur

[Group 2009] GROUP, Object M.: *The Object Management Group (OMG)*. 2009. – URL http://www.omg.org/

[Kecher 2005] KECHER, Christoph: *UML 2.0. Das umfassende Handbuch*. Galileo Press, 2005. – URL http://www.amazon.de/exec/obidos/redirect?tag=citeulike01-21\&path=ASIN/3898427382. – ISBN 3898427382

[Microsoft 2008] MICROSOFT: *Übersicht über das Visio-Objektmodell*. 2008. – URL http://msdn.microsoft.com/de-de/library/cc160740.aspx

[Nantes 2008] NANTES, Universite D.: *The AMMA home page*. 2008. – URL http://www.sciences.univ-nantes.fr/lina/atl/

[Papadopoulos 2008] PAPADOPOULOS, G.A.: Automatic code generation: A practical approach. In: *Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on*, June 2008, S. 861–866. – ISSN 1330-1012

[Pires u. a. 2008] PIRES, Waldemar ; BRUNET, ao ; RAMALHO, Franklin: UML-based design test generation. In: *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*. New York, NY, USA : ACM, 2008, S. 735–740. – ISBN 978-1-59593-753-7

[Shu u. a. 2008] SHU, Chen ; QING, Wu G. ; JING, Xiao: Metamodel Approach on Model Conformance and Multiview Consistency Checking. In: *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM '08. 4th International Conference on*, Oct. 2008, S. 1–4

[Stahl u. a. 2007] STAHL, Thomas ; VÖLTER, Markus ; EFFTINGE, Sven: *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. Dpunkt Verlag, 2007. – URL http://www.amazon.de/exec/obidos/redirect?tag=citeulike01-21\&path=ASIN/3898644480. – ISBN 3898644480

[Sutton und Maletic 2005] SUTTON, Andrew ; MALETIC, Jonathan I.: Mappings for Accurately Reverse Engineering UML Class Models from C++. In: *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*. Washington, DC, USA : IEEE Computer Society, 2005, S. 175–184. – ISBN 0-7695-2474-5

[Vinita u. a. 2008] VINITA ; JAIN, Amita ; TAYAL, Devendra K.: On reverse engineering an object-oriented code into UML class diagrams incorporating extensible mechanisms. In: *SIGSOFT Softw. Eng. Notes* 33 (2008), Nr. 5, S. 1–9. – ISSN 0163-5948

[W3C 1999] W3C: *XML Path Language (XPath) Version 1.0*. 1999. – URL http://www.w3.org/TR/xpath

[Waldén und Nerson 1995] WALDÉN, Kim ; NERSON, Jean-Marie: *Seamless object-oriented software architecture: analysis and design of reliable systems*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1995. – ISBN 0-13-031303-3

Vinita u. a. (2008) Pires u. a. (2008) Shu u. a. (2008) Papadopoulos (2008)

Sutton und Maletic (2005) Waldén und Nerson (1995) Stahl u. a. (2007) Kecher (2005) Group (2009)

Microsoft (2008) W3C (1999) Nantes (2008)