Verhaltensbasierte Architekturkonformitätsüberprüfung[1]

Ana Nicolaescu[2]

Abstract: Architekturkonformitätsüberprüfung ist wichtig, um die unvermeidliche Abweichung zwischen der vorgesehenen Architekturbeschreibung und der eigentlich implementierten Architektur eines Softwaresystems zu kontrollieren. In den letzten Jahren wurden einige Ansätze für eine statisch-basierte Architekturkonformitätsüberprüfung vorgeschlagen. Diese haben jedoch erhebliche Nachteile, insbesondere dann, wenn damit moderne, technologisch heterogene und verteilte Systeme analysiert werden sollen. Eine Prüfung der Architekturkonformität solcher Systeme kann häufig nicht mit statisch-basierten Verfahren durchgeführt werden, da diese nicht immer feststellen können, ob sich ein System so verhält, wie dies vorgesehen ist. In dieser Arbeit stellen wir ARAMIS vor, einen verhaltensbasierten Ansatz zur Prüfung der Architekturkonformität, der dieses Problem löst.

1 Einleitung

Änderungen an Softwaresystemen werden normalerweise unter Zeit- und Kostendruck durchgeführt und sind daher nicht oder nur unzureichend dokumentiert. Sehr häufig verletzen diese auch die ursprünglichen Architekturentscheidungen zugunsten von weniger adäquaten, aber kurzfristig einfacher zu implementierenden Lösungen [Ga13], [dSB12]. Diese Tatsache mag kurzfristig akzeptabel sein. Wenn aber mittel- und langfristig keine korrigierenden Maßnahmen ergriffen werden, entwickelt sich das System sehr schnell anders als in der vorgesehenen Architekturbeschreibung [PW92], [LV95] vorgegeben. Dadurch wird die vorgesehene Architekturbeschreibung allmählich unbrauchbar oder sogar schädlich für die Unterstützung des Systemverständnisses auf einer abstrakteren, konzeptionelleren Ebene. Infolgedessen wurde eine Vielzahl von Ansätzen zum Extrahieren implementierter Architekturbeschreibungen vorgeschlagen ([DP09], [KP07]). Die meisten basieren jedoch auf Strukturanalysen der Artefakten eines Systems (Quellcode, Konfigurationsdateien usw.) und geben daher nur die statische Sicht des Systems wider. Die Komplexität moderner Systeme beruht jedoch häufig auf dem Zusammenspiel verschiedener Subsysteme und nicht nur auf ihrer Struktur [Si10].

ARAMIS ist unser verhaltensbasierter Ansatz zur Lösung des oben genannten Problems [Ni10].In einem ersten Schritt wird dabei die vorgesehene Architekturbeschreibung des zu analysierenden Softwaresystems mittels eines ARAMIS-spezifischen Metamodells erfasst. Diese Beschreibung besteht im Wesentlichen aus den Architektureinheiten des Systems und ihren Kommunikationsregeln. Um die Akzeptanz des Ansatzes zu erhöhen, werden darüber hinaus Techniken des Modell-Engineerings genutzt. Dabei wird das Ziel verfolgt, schon existierende Architekturbeschreibungen verarbeiten zu können, die nicht dem

[1] Behavior-Based Architecture Conformance Checking

[2] Research Group Software Construction, nicolaescu@swc.rwth-aachen.de

ARAMIS-Metamodell entsprechen. Im Anschluss daran werden die Interaktionen innerhalb einer ausgeführten Software aufgezeichnet, wobei vorhandene Monitoring-Systeme verwendet werden. Im Gegensatz zu den statisch-basierten Ansätzen, ist eine vollständige Analyse eines nicht trivialen Softwaresystems jedoch unmöglich. Um dem entgegenzuwirken, werden eine Reihe von Indikatoren eingeführt, die Hinweise bezüglich der Angemessenheit des analysierten Verhaltens, im Verhältnis zu einer systemweiten Konformitätsüberprüfung, liefern. Die Interaktionen innerhalb des Systems werden im nächsten Schritt analysiert. Dabei wird die Kommunikation den definierten Architektureinheiten zugewiesen und anschließend gegen die definierten Kommunikationsregeln geprüft. Dieses Ergebnis liefert die implementierte Architektur des Softwaresystems, die insbesondere auch die Abweichungen von der vorgesehenen Architekturbeschreibung definiert. Die implementierte Architektur kann vielfältig analysiert werden, so können zum Beispiel benutzerdefinierte Architektursichten und Perspektiven definiert oder dedizierte Visualisierungen genutzt werden. Abschließend werden Prozesse vorgeschlagen, die einen Leitfaden für eine verhaltensbasierte Architekturkonformitätsüberprüfung darstellen.

In Anbetracht der Beschränkungen hinsichtlich des Umfangs dieser Veröffentlichung werden wir in den nächsten Abschnitten nur eine Teilmenge unserer Ergebnisse präsentieren und den Leser für weitere Details auf die vollständige Dissertation verweisen cite nicolaescuDis. Zunächst geben wir einen Überblick über das Metamodell der ARAMIS-Architekturbeschreibungen und die Regeln, die mit der ARAMIS-Regelsprache ausgedrückt werden können. Wir fahren mit einem Überblick über die Indikatoren fort, die entwickelt wurden, um die Angemessenheit eines extrahierten Verhaltens zu analysieren, eine verhaltensbasierte Architekturkonformitätsprüfung zu unterstützen. Zuletzt stellen wir das im Rahmen unserer Forschung identifizierte Metamodell-Inkompatibilitätsproblem vor, geben einen kurzen Überblick über unsere Lösung und schließen diese Arbeit mit einer Diskussion über unsere Ergebnisse und die identifizierten Validitätsbedenken ab.

2 ARAMIS Architekturbeschreibungen

Da ARAMIS eine verhaltensbasierte Analyse einsetzt, muss das System überwacht und die auftretenden Interaktionen aufgenommen werden, da diese die Beziehungen zwischen den Codebausteinen des Systems abbilden. Dem hierarchischen Reflexionsmodell [KS03] entsprechend, werden diese extrahierten Beziehungen folglich auf die Elemente der beabsichtigten Architekturbeschreibung angewendet. Folglich werden damit die Konvergenzen, Divergenzen und Abwesenheiten identifiziert. In den nächsten Abschnitten stellen wir das Metamodell der ARAMIS Architekturbeschreibungen vor.

Wie im Metamodell in Abbildung 1 dargestellt, hat eine Architekturbeschreibung je nach ihrer Beziehung zum betreffenden Softwaresystem entweder eine präskriptive oder deskriptive Rolle. In ihrer präskriptiven Rolle wird die Beschreibung als die vorgesehene Architekturbeschreibung des Systems (*intended architecture description*) bezeichnet. Umgekehrt wird die Beschreibung in ihrer deskriptiven Rolle - in der sie hauptsächlich darstellt, wie das System tatsächlich aufgebaut ist - als die Beschreibung der implementierten Architektur des Systems (*implemented architecture description*) bezeichnet.

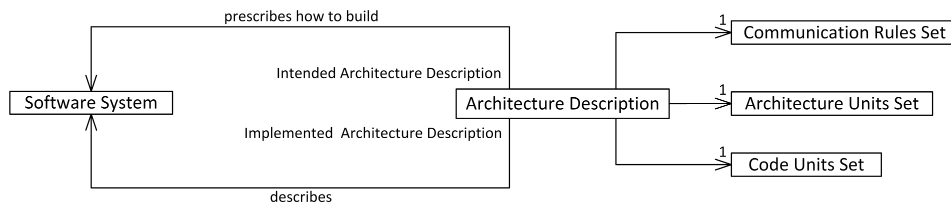

Abb. 1: ARAMIS Architekturbeschreibungen

Unabhängig von ihrer Rolle bestehen Architekturbeschreibungen aus drei Mengen: die Menge der Architektureinheiten, die Menge der Codeeinheiten und die Menge der Kommunikationsregeln. Die Menge der Codeeinheiten (*code units set*) beinhaltet alle im Rahmen einer Architekturbeschreibung definierten Codeeinheiten. Eine Codeeinheit (*code unit*) ist eine nicht-typisierte, programmiersprachenunabhängige Abstraktion eines konkreten Codebausteins.

Wie im Metamodellausschnitt in Abbildung 2 dargestellt, haben wir Codeeinheiten als eigenständige atomare Elemente modelliert. Auch wenn die Codeeinheit einen Codebaustein (z.B. Paket) darstellt, der selbst aus weiteren Codebausteinen (z.B. Klassen) besteht, enthält die Codeeinheit keine weiteren Codeeinheiten, sondern gilt als Platzhalter für alle betroffenen Bausteine. Dadurch wurde das Meta-Modell einfach gehalten und der Modellierungsaufwand wurde reduziert.

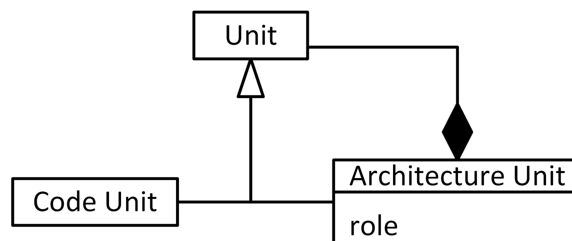

Abb. 2: Architecture Units in ARAMIS

Die Menge der Architektureinheiten (*architecture units set*) beinhaltet alle im Kontext der Architekturbeschreibung eines Systems definierten Architektureinheiten. Eine Architektureinheit (*architecture unit*) fasst Teile eines Softwaresystems zusammen, die gemeinsam eine architektonischen Bedeutung aufweisen. Eine Architektureinheit spiegelt sich nicht unbedingt explizit im Code wider. Eine Architektureinheit kann aus Codeeinheiten und weiteren Architektureinheiten bestehen. Folglich, wie in Abbildung 2 dargestellt, definiert eine Architektureinheit ein hierarchisches Konstrukt mit einem eindeutigen Identifikator. Das Architektureinheitskonzept hat eine lose Semantik. Dies steht im Gegensatz zu den meisten anderen Architekturbeschreibungssprachen, die in verwandten Arbeiten definiert sind. Der Grund dafür war, Flexibilität zu ermöglichen, da Architekten kein Standard-Meta-Modell zur Beschreibung von Architekturen verwenden. Z.B. bestehen in einigen Beschreibungen Schichten aus Komponenten, in anderen wiederum sind Komponenten

selbst geschichtet etc. Nicht zuletzt unterstützt eine lose Semantik auch die Möglichkeit, mehrere Ansichten der Architekturbeschreibung zu modellieren. Mit Blick auf das 4+1-Modell von Kruchten [Kr95], können die Architektureinheiten somit flexibel eingesetzt werden, um z.B. die Konstrukte der logischen Sicht des Systems zu modellieren, aber auch um die Prozesse in ihrer Prozesssicht darzustellen, indem Architektureinheiten für die während der Laufzeit existierenden Prozesse definiert werden.

Die Menge der Kommunikationsregeln (*communication rules set*) fasst alle Kommunikationsregeln zusammen, die die Kommunikation von Architektureinheiten regeln und werden im folgenden Abschnitt näher erläutert.

3 Kommunikationsregeln

Konformitätsprüfungsansätze sollten über triviale strukturelle Untersuchungen hinausgehen. Architekten äußerten oft die Notwendigkeit, Kommunikationsregeln für komplexe Systeme formulieren zu können. Diese Systeme weisen z.B. komplexe Interaktionsmuster und Kommunikationsmechanismen auf, die vor der Laufzeit nicht analysierbar sind. Bei der Gestaltung der Regelspezifikationssprache von ARAMIS war es unser Ziel, sicherzustellen, dass auch Regeln formuliert werden können, die z.B. Einschränkungen der verwendeten Kommunikationsprotokolle sowie indirekte Kopplungen und asynchrone Interaktionen ausdrücken können. Wie in [Ni10] ausführlich beschrieben, baut ARAMIS auf bestehenden Softwaremonitoren auf, um Interaktionen aus einem laufenden System zu extrahieren. Eine Interaktion wird durch ihren Aufrufer (*caller*), Aufgerufenen (*callee*) und eine Menge von Kommunikationsparametern (*communication parameters*) gekennzeichnet. Die Kommunikationsparameter stellen weitere technische Details der Kommunikation dar (z.B. das verwendete Kommunikationsprotokoll). ARAMIS verwendet dann einen auf Reguläräusdrücken basierten Ansatz, um den Aufrufer und den Aufgerufenen auf entsprechenden Architektureinheiten abzubilden. Auf diese Weise entstehen mehrstufige Abstraktionen des analysierten Verhaltens. Die architektonisch abgebildeten Interaktionen können dann mit Hilfe von Kommunikationsregeln validiert werden.

Die Kommunikationsregeln stehen im Mittelpunkt der ARAMIS-Analyse. Eine taxonomische Übersicht der ARAMIS-Kommunikationsregeln ist in Abbildung 3 dargestellt. Erstens kann eine Regel je nach Berechtigungstyp eine bestimmte Kommunikation erlauben (*allow*), verweigern (*deny*) oder erzwingen (*enforce*). Darüber hinaus können die Regeln je nach Herkunft entweder spezifiziert (*specified*) (z.B. Einheit A kann Einheit B aufrufen), abgeleitet (*derived*) (Einheit A kann Einheit B aufrufen, da Einheit A in Einheit X enthalten ist und Einheit B in Einheit Y enthalten ist und es spezifiziert wurde, dass X Y aufrufen darf) oder default sein. Default Regeln regeln die Kommunikation für die Fälle, in denen keine spezifizierten oder abgeleiteten Regeln zutreffen. Darüber hinaus unterscheiden wir je nach Kommunikationsart zwischen (1) Caller-Callee Regeln, die die gerichtete Kommunikation zwischen einem Paar spezifizierter Caller- und Callee-Architektureinheiten betreffen (z.B. Schicht A darf nicht auf Schicht B zugreifen), (2) Caller-Regeln, die die von einer Einheit zu allen anderen ausgehende Kommunikation betreffen (z.B., die *utilities* Architektureinheit darf keine anderen Architektureinheiten auf-

rufen) und (3) Callee-Regeln, die sich auf die Kommunikation beziehen, die auf eine bestimmte Callee-Architektureinheit abzielt (z.B. die Fassade Schicht kann von allen anderen Architektureinheiten aufgerufen werden). Darüber hinaus können Regeln mit einem

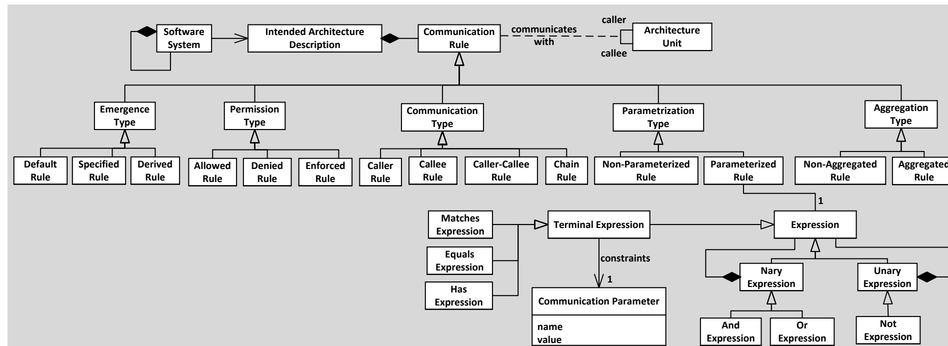

Abb. 3: Taxonomie der ARAMIS Regeln

Parameter versehen werden, um Beschränkungen für beliebigen Kommunikationsparameter zu spezifizieren. Das zugrunde liegende Modell der Kommunikationsparameter ist von dem für die Extraktion verwendeten Überwachungswerkzeug abhängig. In einem ersten Schritt werden die Interaktionen durch ARAMIS-Adapter extrahiert. Die dazugehörenden Kommunikationsparameter werden in entsprechende Schlüssel-Wert-Paare transformiert. Solche Schlüssel können z.B. die Art der Kommunikation darstellen (z.B. Queue, SOAP usw.). Folglich können Regeln eine Kommunikation erlauben oder verweigern, indem sie auf mehrere Parameter verweisen, die durch logische Operatoren verbunden sind, wie z.B. oder (*or*), und (*and*) oder nicht (*not*) (z.B. die Kommunikation zwischen zwei Architektureinheiten ist nur erlaubt, wenn die erste auf die zweite über einen restful Web-Service zugreift und wenn der Name des aufgerufenen Endpunktes mit einem bestimmten regulären Ausdruck validiert werden kann). Um die Spezifikation solcher Regeln zu ermöglichen, haben wir eine ausdruckbasierte Sprache definiert, die aus drei Arten von Ausdrücken besteht: Terminalausdrücke (*terminal expressions*) schränken die Kommunikationsparameter direkt ein. Z.B. können die Ausdrücke *matches* und *equals* verwendet werden, um eine Kommunikation zuzulassen/zu verweigern, wenn der Wert eines Kommunikationsparameters (z.B. "Queue-Name") mit einem bestimmten regulären Ausdruck übereinstimmt (z.B. <key: Warteschlangenname, Wert: matches (queueData*)>) oder einen bestimmten Wert hat (z.B. <key: Warteschlangenname, Wert: equals (queueDataTransfer)>).

N-äre (*n-ary*) und unäre (*unary*) Ausdrücke können verwendet werden, um Bedingungen durch logische Operatoren auszudrücken. Der unäre nicht (*not*) kann auf einen Ausdruck angewendet werden, um seinen booleschen Wert zu invertieren. Darüber hinaus kann man unter Verwendung eines n-ären Ausdrucks eine Kommunikation einschränken, wenn (1) mehrere ausdruckbasierte Bedingungen gleichzeitig gelten (*und* Ausdrücke—z.B. der Kommunikationstyp sollte SOAP Webservice sein und der Endpunktname sollte mit dem regulären Ausdruck "getData*"übereinstimmen) oder (2) wenn einige von ihnen gelten sollen(*oder* Ausdrücke—z.B., die Kommunikation ist erlaubt, wenn die Art der Kommunikation SOAP Webservice oder restful Webservice ist).

Darüber hinaus können wir je nach Komplexität der Kommunikationsregel zwischen aggregierenden (*aggregating*) und nicht-aggregierenden (*non-aggregating*) Regeln unterscheiden. Die nicht-aggregierenden Regeln entsprechen Konformitätsüberprüfungen, die für einzelne Interaktionen durchgeführt werden. So kann beispielsweise eine nicht aggregierende Regel voraussehen, dass Einheit A nicht auf Einheit B zugreifen sollte. Folglich kann im Falle einer architektonisch abgebildeten Interaktion diese sofort gegen diese Regel validiert werden, da keine zusätzlichen Informationen über andere Interaktionen benötigt werden. Bei einer aggregierenden Regel müssen mehrere Interaktionen berücksichtigt werden. Zwei Beispiele für solche Regeln sind (1) Einheit A sollte mit Einheit B über die Datenbank gekoppelt werden und (2) Einheit A sollte mit Einheit B über einen einzigen Kommunikationsmechanismus interagieren.

Dadurch ermöglicht ARAMIS die Definition sehr spezifischer Regeln, die die vorgesehene Kommunikation innerhalb eines Systems einschränken. Zu diesem Zweck wurde die XML-basierte Aramis-Regelsprache eingeführt, wie in [Ni10] beschrieben.

4 Untersuchung der Angemessenheit

Um fundierte Ergebnisse einer verhaltensbasierten Architektur-Konformitätsprüfung zu erhalten, muss auch sichergestellt werden, dass die Überwachung auf einer angemessenen Grundlage durchgeführt wird. Zu diesem Zweck haben wir eine Reihe von White-Box und Black-Box-Indikatoren vorgeschlagen. White-Box-Indikatoren geben eine Einschätzung darüber ab, inwieweit ein System aus der Sicht seiner vorgesehenen und implementierten Architekturen überwacht wurde. Im Gegensatz dazu analysieren die Black-Box-Indikatoren das Ausmaß, in dem ein System hinsichtlich seiner Spezifikation untersucht wurde.

4.1 White-Box Indikatoren

Auf einer ersten Analyse-Ebene kann mit jedem bekannten Codeabdeckung Metrik abgeschätzt werden, inwieweit ein System untersucht wurde. Eine hohe Codeabdeckung kann jedoch relativ einfach auf der Unit Tests-Ebene erreicht werden. Nichtdestotrotz argumentieren wir, dass man das System als Ganzes betrachten muss, wenn das Verhalten bezüglich Architekturkonformitätsüberprüfungen untersucht wird. Auf dieser Integrationsebene ist es oft schwierig, eine hohe Abdeckung zu erreichen. Zu diesem Zweck haben wir ([Ni10]) eine Reihe von Indikatoren definiert, um die folgenden Fragen zu beantworten:
Q1. Inwieweit wurde das System als Ganzes abgedeckt?

Q2. Gibt es Einheiten, die nicht abgedeckt wurden, obwohl sie in der vorgesehenen Architekturbeschreibung definiert wurden?

Q3. Inwieweit wurde eine bestimmte Einheit abgedeckt?

Q4. Inwieweit wurde die vorgesehene Kommunikation während der Überwachung ausgelöst?

Die vorgeschlagenen Indikatoren, die die oben genannten Fragen unterstützen, können dann top-down oder bottom-up analysiert werden. Wenn eine niedrige bekannte Codeabdeckung (z.B. Anweisungsabdeckung) erreicht wird, ist der Architekt möglichst skeptisch, was die Angemessenheit der durchgeführten Überwachung betrifft. Einerseits könnte er durch die top-down-Untersuchung der vorgesehenen Architektur feststellen, ob es Architektureinheiten gibt, die bei der Überwachung nicht abgedeckt wurden. Eine top-down-Traversierung der Architektureinheitenhierarchie könnte zunächst hochrangige nicht abgedeckte Architektureinheiten aufdecken. Nach der Identifizierung einer nicht oder schlecht abgedeckten Architektureinheit kann der Architekt Annahmen darüber treffen, warum diese Einheit nicht (richtig) in das überwachte Verhalten des Systems involviert war. Z.B. kann er erkennen, welches Verhalten ausgelöst werden muss, um die Beteiligung der identifizierten Architektureinheit zu erhöhen. Der Architekt kann dann seine Annahmen verfeinern, indem er die Abdeckung der tieferen Architektureinheiten genauer analysiert. Zu diesem Zweck kann er z.B. untersuchen, ob alle darin enthaltenen Codeeinheiten während der Überwachung verwendet wurden. Bei Bedarf kann er durch die Untersuchung der bekannten Codeabdeckungsmetriken der Codeeinheiten ein technisch besseres Verständnis über den Umfang der durchgeführten Überwachung erlangen. Zu jedem Zeitpunkt der Analyse kann der Architekt auch untersuchen, welche Kommunikationsregeln bei der Architekturkonformitätsprüfung verwendet wurden. Diese könnten zusätzlich die Identifizierung von Verhaltensvarianten unterstützen, die bei der Überwachung nicht ausgelöst wurden. Diese können wiederum Hinweise zur Verbesserung der Angemessenheit geben. Ergibt die zuvor vorgestellte Analyse keine wichtigen Abwesenheiten, kann der Architekt das erfasste Verhalten und die durchgeführte Konformitätsprüfung trotz des möglicherweise niedrigen Wertes der bekannten Codeabdeckungsmetriken als angemessen darstellen.

4.2 Black-Box Indikator

Unsere durch Fallstudien bestätigte Annahme ist, dass das extrahierte Verhalten eines Systems eine adäquate Grundlage für eine verhaltensbasierte Architekturkonformitätsprüfung darstellt, wenn es so viele relevante Szenarien des Systems wie möglich umfasst und wenn diese in ihren unterschiedlichsten Kontexten durchgeführt werden. In diesem Zusammenhang ist ein Szenario relevant, wenn es einer Schlüsselanforderung entspricht oder wenn es vom Architekten als besonders geeignet erachtet wird, das Verhalten des Systems über seine Architektureinheiten hinweg darzustellen. In [Ni10] haben wir die *Scenario Coverage Metric* definiert, um eine Abschätzung über die Angemessenheit eines extrahierten Verhaltens im Hinblick auf die Szenarien des Systems zu geben. Diese Metrik hat sich in unseren durchgeführten Fallstudien als nützlich erwiesen und basiert hauptsächlich auf den folgenden wichtigen Annahmen:

1. Die in nicht relevanten Szenarien auftretenden Architekturabweichungen haben weniger Einfluss auf die architektonische Gesamtqualität eines Systems. Die Wahrscheinlichkeit, den entsprechenden Code zu entwickeln, ist geringer: es ist wahrscheinlicher, dass die Systemteile, die den relevanten Szenarien entsprechen, weiterentwickelt werden, da neue oder sich ändernde Anforderungen oft durch ständige Nutzung ausgelöst werden.

2. Es gibt verschiedene Kontexte, in denen ein Szenario ausgeführt werden kann. Im Rahmen einer Architekturkonformitätsprüfung können diese unterschiedliche Ergebnisse liefern.

5 Das Meta-Modell-Inkompatibilitätsproblem

Es gibt keinen Konsens bezüglich einer universellen Sprache zur Formulierung vorgesehener Architekturbeschreibungen. Wie Malavolta betonte, würde eine solche universelle Sprache wahrscheinlich nicht einmal an Popularität gewinnen [Ma13]. Die Vielfalt der Möglichkeiten, Architekturbeschreibungen zu formulieren, spiegelt sich auch in den verfügbaren Werkzeugen zur Konformitätsprüfung wider. Diese verwenden proprietäre Metamodelle, die oft nicht erweiterbar sind und eine spezifische Semantik aufweisen. Das *Meta-Modell-Inkompatibilitätsproblem* drückt die syntaktische und/oder semantische Uneinigkeit zwischen den Sprachen aus, die einerseits von den Architekten bei der Erstellung von Architekturbeschreibungen verwendet werden und die andererseits von den verschiedenen Tools zur Architekturkonformitätsüberprüfung eingesetzt werden, um vorgesehene und/oder implementierte Architekturen darzustellen. Während das ARAMIS Meta-Modell für Architekturbeschreibungen nur sehr wenige semantische Einschränkungen aufweist, bleibt das Problem der Meta-Modell-Inkompatibilität bestehen. Wenn ein beliebiges System mit ARAMIS analysiert wird, stehen die Chancen gut, dass seine vorgesehene Architekturbeschreibung mit einem anderen Metamodell als ARAMIS definiert wurde.

Um die Einschränkung durch das Meta-Modell-Inkompatibilitätsproblem im Rahmen von ARAMIS zu verringern, haben wir den ARAMIS Architecture Description Transformation Process (AADT-Proc) entwickelt. Der AADT-Proc unterstützt die Transformation von ursprünglich vorgesehenen Beschreibungen zu ARAMIS-spezifischen Äquivalenten. Der AADT-Proc kann dazu dienen, die Architekten im Falle einer manuellen Transformation anzuleiten oder die Entwicklung von automatischen Transformationen zu unterstützen. Das Ziel von automatischen Transformationen ist, flexible Formate für die vorgesehene und implementierte Architekturbeschreibung zu ermöglichen. So könnte dann ein Architekt eine non-ARAMIS vorgesehene Architekturbeschreibung als Input für die ARAMIS-Architekturkonformitätsüberprüfung liefern und als Output das gleiche Diagramm erhalten, das aber durch zusätzliche Ergebnisse erweitert wird. Der AADT-Proc wird ausführlich in [Ni10] vorgestellt.

6 Diskussion

Die verhaltensbasierte Architekturkonformitätsprüfung ist in der Regel teurer als die statische Variante, da sie die Extraktion von Interaktionen aus laufenden Systemen und deren anschließende Analyse voraussetzt. Trotz dieser Einschränkung erweist sich eine verhaltensbasierte Architekturkonformitätsprüfung als nützlich, wenn man moderne Systeme betrachtet. Dieser Aspekt wurde in [Ni10] ausführlich diskutiert, sowohl auf theoretischer als auch auf praktischer Ebene. Darüber hinaus kann die Komplexität der verhaltensbasierten Architekturkonformitätsprüfung reduziert werden, wenn der verwendete Ansatz

auf existierende Performance Monitoring-Systemen aufbaut, die es bereits ermöglichen, Interaktionen aus einer Vielzahl von Systemen zu extrahieren.

Des Weiteren haben wir untersucht, welche Arten von Regeln ausdrückbar sein sollten, um das vorgesehene Verhalten eines bestimmten Systems auf einer architektonischen Ebene zu charakterisieren. Die vorgeschlagene Taxonomie war ausreichend, um die Definition aller Kommunikationsregeln zu unterstützen, die in den durchgeführten Fallstudien identifiziert wurden. Einer der wichtigsten in diesem Zusammenhang gelernten Aspekte war die Vermeidung unflexibler Lösungen. Folglich können mit Hilfe unserer Regeln Beschränkungen für jeden während der Überwachung extrahierten Interaktionsparameter festgelegt werden.

Eine weitere wichtige Erkenntnis ist, dass Kommunikation eine wichtige Rolle bei der Akzeptanz der Konformitätsergebnisse spielt. Mit unserer Model-Engineering-basierten Lösung des Meta-Modell-Inkompatibilitätsproblems erzielten wir eine Erhöhung der Akzeptanz von ARAMIS. In einer der drei durchgeführten Fallstudien lehnten aber die Architekten unsere Model-Engineering Lösung ab. Sie argumentierten, dass eine augmentierte Version der vorgesehen Architektur unübersichtlich wäre. Anstatt dessen sei eine einfache tabellarische Darstellung der Ergebnisse vorteilhafter. Die Kommunikation mit den Stakeholdern und das Verständnis ihrer Bedürfnisse können also den investierten Aufwand erheblich reduzieren.

Nicht zuletzt haben wir gelernt, dass die Codeabdeckung nicht immer eine gute Schätzung der Angemessenheit des erfassten Verhaltens darstellt. In den durchgeführten Fallstudien haben wir gezeigt, dass selbst bei geringer Codeabdeckung das extrahierte Verhalten ausreichend sein kann, um eine verhaltensbasierte Architekturkonformitätsprüfung zu unterstützen. Die Angemessenheit des Verhaltens kann stattdessen effektiver untersucht werden, indem man eine Reihe von vorgeschlagenen White-Box- und Black-Box-Indikatoren entsprechend einsetzt.

Einschränkungen

Dieser Abschnitt gibt einen Überblick der wichtigsten Einschränkungen unserer Arbeit.

ARAMIS baut auf bestehenden Monitoren auf, aber diese sind nicht für ihren Zweck optimiert. Eines der Ziele von ARAMIS war es, bereits bestehende Arbeiten im Bereich Systemüberwachung wiederzuverwenden. Dies stellt aber gleichzeitig eine Einschränkung dar. Die meisten Monitore sind für die Identifizierung von Performance-Problemen optimiert. Dadurch werden erhebliche Datenmengen extrahiert, die zu Schwierigkeiten in der ARAMIS Analyse führen. Stattdessen könnten kundenspezifische, auf den Architekturbeschreibungen basierende Instrumentierungen dieses Problem weitgehend lindern.

ARAMIS identifiziert Architekturverletzungen, kann diese aber nicht nach Priorität einordnen. Die Identifizierung auftretender Verletzungen ist eine wichtige Aktivität,

die die Bewertung und Weiterentwicklung der Architektur unterstützt. Eine große Anzahl von festgestellten Verstößen kann jedoch zu Demoralisierung und Zurückhaltung bei der Verbesserung des Systems führen. Derzeit werden alle durch ARAMIS identifizierten Verstöße gleich behandelt. Eine automatische Priorisierung nach flexiblen Kriterien, wie z.B. Performance-Auswirkungen oder betroffenes architektonisches Niveau, könnte die beteiligten Architekten weiter motivieren, die identifizierte architektonische Abweichung schrittweise zu reduzieren.

References

[DP09] Ducasse, Stephane; Pollet, Damien: Software Architecture Reconstruction: A Process-Oriented Taxonomy. IEEE Transactions on Software Engineering, 35(4):573–591, 2009.

[dSB12] de Silva, Lakshitha; Balasubramaniam, Dharini: Controlling Software Architecture Erosion: A Survey. Journal of Systems and Software, 85(1):132–151, January 2012.

[Ga13] Garcia, Joshua; Krka, Ivo; Mattmann, Chris; Medvidovic, Nenad: Obtaining Ground-truth Software Architectures. In: Proc.of the International Conference on Software Engineering (ICSE). IEEE Press, Piscataway, NJ, USA, S. 901–910, May 2013.

[KP07] Knodel, Jens; Popescu, Daniel: A Comparison of Static Architecture Compliance Checking Approaches. In: 6th Working IEEE/IFIP Conference on Software Architecture (WICSA), Mumbai, Maharashtra, India. S. 12–21, January 2007.

[Kr95] Kruchten, Philippe: The 4+1 View Model of Architecture. IEEE Software, 12(6):42–50, November 1995.

[KS03] Koschke, Rainer; Simon, Daniel: Hierarchical Reflexion Models. In: Proceedings of the 10th Working Conference on Reverse Engineering (WCRE). IEEE, S. 36–45, 2003.

[LV95] Luckham, David C.; Vera, James: An Event-Based Architecture Definition Language. IEEE Transactions on Software Engineering, 21(9):717–734, September 1995.

[Ma13] Malavolta, Ivano; Lago, Patricia; Muccini, Henry; Pelliccione, Patrizio; Tang, Antony: What Industry Needs from Architectural Languages: A Survey. IEEE Transactions on Software Engineering, 39(6):869–891, jun 2013.

[Ni10] Nicolaescu, Ana: Behavior-Based Architecture Conformance Checking. Shaker, 1st. Auflage, 2010.

[PW92] Perry, Dewayne E.; Wolf, Alexander L.: Foundations for the Study of Software Architecture. SIGSOFT Softw. Eng. Notes, 17(4):40–52, Oktober 1992.

[Si10] Sigelman, Benjamin H.; Barroso, Luiz Andr; Burrows, Mike; Stephenson, Pat; Plakal, Manoj; Beaver, Donald; Jaspan, Saul; Shanbhag, Chandan: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Bericht, Google, Inc., 2010.

Ana Nicolaescu verteidigte ihre Doktorarbeit im September 2018 an der RWTH Aachen University. Ihre Kernkompetenzen liegen im Software-Engineering-Bereich mit dem Schwerpunkt Software-Architektur. Sie hat zuvor ihr Master-Studium an der RWTH Aachen und ihr Bachelor-Studium an der Politehnica University of Bukarest abgeschlossen.