Sicherstellung der Funktionalität in Komponentensystemen und Service-orientierten Architekturen.

Andreas Both und Wolf Zimmermann
Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg
{andreas.both,wolf.zimmermann}@informatik.uni-halle.de

Abstract: Komponentensysteme besitzen ebenso wie Service-orientierte Architekturen große Vorteile bei der Erstellung von Anwendungen aus vielen beziehungsweise unbekannten Komponenten (Services). Service-orientierte Architekturen werden häufig genutzt, um Arbeitsabläufe zu integrieren. Leider ist es nicht möglich, die Errungenschaften bei der Sicherstellung von Funktionalität von imperativen beziehungsweise objektorientierten Programmen – z. B. Optimierungen, Vorhersage von Verhalten etc. – zu integrieren. Da die Komponenten häufig unbekannt sind und der Quellcode nicht zugreifbar ist, fehlt einerseits eine Möglichkeit des Vorhersagens von dem Verhalten einer Komponente. Andererseits ist es nur sehr schwer möglich, Anwendungen zu implementieren, deren Funktionalität bereits bei der Konfiguration der Komponenten abgesichert werden kann. Darüber hinaus implementieren Komponenten häufig einen eigenen Geschäftsprozess. Theoretisch ist die Vorhersage des Verhaltens nur bei zustandlosen Komponenten mit exakt einer Schnittstelle problemlos möglich, praktisch resultiert aus diesen Eigenschaften ein erhöhter Testaufwand. Die Funktionalität von Komponentensystemen und Service-orientierten Architekturen kann also nicht auf traditionelle Art und Weise sichergestellt werden. In dieser Arbeit stellen wir einen Ansatz vor, der sicherstellen kann, dass die vorgesehenen Workflows eingehalten werden.

1 Motivation

Die zunehmende Verbreitung von Komponentensystemen und Service-orientierten Architekturen verstärkt den Trend in der Softwareentwicklung in zwei Richtungen. Die benötigten Bausteine werden im zunehmenden Maße nicht mehr innerhalb eines (einzigen) Entwicklungshauses erzeugt. Sie werden häufiger als früher von externen Firmen entwickelt beziehungsweise eingekauft oder es wird eine existierende ausgelieferte Komponente verwendet (auch dynamisch eingebunden). Des Weiteren ist ein starker Trend zu „kooperativen“ Anwendungen beobachtbar. Dies sind Anwendungen, die Services verschiedener Partner nutzen, um z. B. einen firmenübergreifenden Geschäftsprozess (B2B) zu implementieren.

Darüber hinaus müssen Softwareentwickler heutzutage mit Komponenten arbeiten, die häufig vor langer Zeit implementiert wurden und vor der Integration mühsam evaluiert werden müssen. Die Situation verschärft sich, wenn die ursprünglichen Entwickler nicht mehr verfügbar sind oder die Komponente in einer dem Entwickler unbekannten Programmiersprache implementiert wurde.

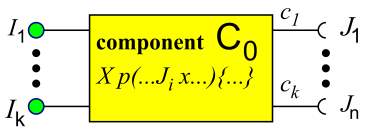

Abbildung 1: Verwendetes Service-Modell.

Ein besonderer Fall sind Komponenten/Services, deren Verhalten nicht evaluierbar ist. Die Ursache kann z. B. darin liegen, dass es sich um einen Webservice handelt. In diesem Fall kann nur mittels der Dokumentation auf die Funktionalität beziehungsweise das Verhalten geschlossen werden.

Grundsätzlich unterscheidet man drei Möglichkeiten der Auslieferung von Komponenten:

- Whitebox: Der Quellcode der Komponente steht zur Verfügung und kann sowohl evaluiert als auch angepasst werden.
- Glassbox: Der Quellcode steht zur Verfügung, kann aber nicht geändert werden.
- Blackbox: Der Quellcode der Komponente ist verborgen und nicht evaluierbar.[1]

Bei Entwicklungen innerhalb eines Softwarehauses verfügt man im Allgemeinen über den Quellcode und kann somit auf die Funktionalität und das Verhalten einer Komponente schließen. Bei externen Komponenten (im Allgemeinen Blackbox-Komponenten) ist dies nicht möglich.

Der von uns vorgeschlagene Prozess wird hier genutzt, um trotz Blackbox-Semantik Aussagen über die Abhängigkeiten bei der Ausführung von Services/Komponenten zu ermöglichen, die sich auf die Interaktionen auswirken.

Diese Arbeit ist wie folgt strukturiert. Im nachfolgenden Kapitel werden Protokolle, Komponenten/Services und deren Abstraktionen dargestellt. In Kapitel 3 wird der bereits früher definierte Verifikationsprozess vorgestellt, der als Ausgangspunkt für den in Kapitel 4 beschriebenen Ansatz zur Überprüfung von Service-orientierten Architekturen genutzt wird. Abschließend werden in Kapitel 5 verwandte Arbeiten diskutiert und in Kapitel 6 die Ergebnisse dieser Arbeit zusammengefasst.

2 Grundlagen

2.1 Komponenten/Services

Wir gehen hier davon aus, dass eine Komponente beziehungsweise ein Service eine Implementierung von jeder Angebotsschnittstelle (provided interface) ist. Es ist möglich, dass

[1] Als Verschärfung dieses Falles müssen Webservices betrachtet werden, weil dort auch der Binärcode verborgen bleibt.

sie auch Nutzungsschnittstellen[2] (required interfaces) benutzen. Wir schränken weder die benutzte Sprache noch den Ort ein, an dem die Komponente beziehungsweise der Service installiert ist. Wir lehnen uns also an die Definition des UML-Komponentenmodells an. Unsere Annahmen sind in Abbildung 1 dargestellt.

Ohne Beschränkung der Allgemeinheit werden wir im Folgenden davon ausgehen, dass die Anwendung unter Benutzung von Webservices – also aus Blackboxes – implementiert wurde. Dies ist besonders wichtig, wenn ein Service beziehungsweise eine Komponente geschäftskritische Daten oder Prozesse enthält.

Im Folgenden verwenden wir zur vereinfachten Schreibweise nur noch den Terminus Service und betrachten nur noch Service-orientierte Architekturen.

Eine SOA-Anwendung ist zusammengesetzt aus Komponenten, die ausschließlich über Nutzungsschnittstellen beziehungsweise Angebotsschnittstellen kommunizieren. Diese Interfaces bestehen aus einer Menge von Funktionen/Prozeduren. Wir erlauben sowohl synchrone als auch asynchrone Schnittstellen. Aufgerufene synchrone Schnittstellen blockieren die weitere Ausführung des aufrufenden Services solange, bis das Ergebnis von der aufgerufenen Komponente geliefert wird. Im Gegensatz dazu werden asynchrone Schnittstellenaufrufe in einem neuen Thread gestartet und der Aufrufer kann die Ausführung sofort fortsetzen. Auf das Ergebnis der Berechnung wird nicht gewartet.

Außerdem gehen wir davon aus, dass das Interface nicht immer Informationen enthält, die darauf schließen lassen, welche Schnittstellen synchron oder asynchron implementiert wurden.

Darüber hinaus schließen wir Rückaufrufe (callbacks) nicht aus. Ebenso erlauben wir eine Weitergabe von Referenzen über Methodenaufrufe (Referenzparameter).

2.2 Service-Protokolle

Ein Service-Protokoll (kurz Protokoll) beschreibt die erlaubten Aufrufreihenfolgen bei der Benutzung eines Services. Ein Protokoll ist immer an einen Service gebunden.

Protokolle sollen Eigenschaften sichtbar und überprüfbar machen, die aus (IT-)Compliance oder Geschäftsregeln folgen und eine bestimmte Benutzung eines Services vorschreiben.

Als Beispiel für ein Protokoll sei hier ein (stark vereinfachtes) Beispiel aus einer Kooperation mit der GISA GmbH aufgeführt. Der entwickelte Service wurde zur Unterstützung der ursprünglich mittels Papier abgewickelten Erfassung und Freigabe von erbrachten Leistungspositionen für die aufwandsbezogene Faktura entworfen. Unter dem Service wurden verschiedene Funktionalitäten in einem SAP-System zusammen gefasst (die die internen Prozesse bereits genau abbildeten) und in einer Service-orientierten Architektur zur Verfügung gestellt. Wenn eine Anwendung sich nicht an das angegebene Protokoll hält, kann dies zu schwerwiegenden Verletzungen der Compliance führen. Das Beispiel ist in Abbildung 2 dargestellt.[3]

[2] folglich Angebotsschnittstellen anderer Services

[3] Weitere Beispiele können auch in [ZS06, BZ08b, BZ08a, BZ09] gefunden werden.

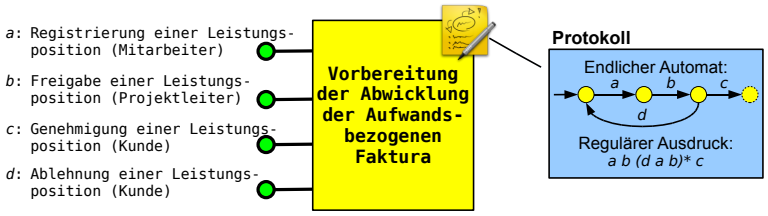

Abbildung 2: Beispiel: Service mit Protokoll (als endlicher Automat und regulärer Ausdruck).

Ein Protokoll kann genutzt werden, um dynamisch eingehende Aufrufe an einem Service zu überprüfen. Diese Überprüfung kann sowohl zur Laufzeit (dynamisch) als auch zur Übersetzungs- beziehungsweise Bindungszeit geschehen. Wir betrachten letzteres, um die Funktionalität von Anwendungen bereits vor der Inbetriebnahme sicherzustellen.

Protokolle werden also an Services gebunden und sichern für genau diese Implementierung eine erlaubte Benutzung – in Form einer Aufrufreihenfolge – ab.

Wir nutzen – in Übereinstimmung mit anderen Arbeiten [ZS06, Reu02, FLNT98] – endliche Automaten, um ein Protokoll zu definieren. Der endliche Automat $P = (Q_P, \Sigma, \rightarrow_P, I_P, F_P)$ ist wie üblich definiert, d. h. Q_P ist eine endliche Menge von Zuständen, Σ ist eine endliche Menge von Aktionen, $\rightarrow_P \subseteq Q_P \times \Sigma \times Q_P$ ist eine endliche Menge von Überführungsregeln, $I_P \in Q_P$ ist der Startzustand, $F_P \subseteq Q_P$ ist die Menge der Finalzustände. Häufig wird auch die äquivalente Darstellung als regulärer Ausdruck genutzt.

Anmerkung 1 *Es ist zu beachten, dass das hier verwendete Protokoll nur eingehende Aufrufe eines Services enthält. Dies ist ein Unterschied zu anderen Arbeiten, z. B. [PP08]. Wir gehen davon aus, dass ein Protokoll flexibler genutzt werden kann, wenn es nur die möglicherweise eingehenden Aufrufe enthält. Darüber hinaus ist es leichter verständlich.*

2.3 Abstraktionen

Wie oben ausgeführt, ist z. B. die Absicherung eines Geschäftsgeheimnisses ein legitimer Grund, den Quellcode eines Services nicht zur Verfügung zu stellen. Verifikationsverfahren müssen damit umgehen können. Aus diesem Grund erstellen wir von jedem Service eine Abstraktion, die das Verhalten dieser Komponente erfasst. Da es sich um eine konservative Abstraktion unter dem Gesichtspunkt der ausgelösten Aufrufe handelt, muss also jeder mögliche Pfad des Kontrollflusses in der Abstraktion vorkommen.

Da Turing-mächtige Abstraktionen nur schlecht während einer Modellprüfung betrachtbar sind und gleichzeitig die grundlegenden Konzepte der Programmiersprachen – Rekursion und Parallelisierung – ermöglicht werden sollen, nutzen wir Process Rewrite Systems zur Darstellung des Verhaltens eines Services.

Process Rewrite Systems [May98] vereinigen Petri-Netze und Kellerautomaten in einem

Modell, indem ein Operator für paralleles Verhalten „||“ und einer für sequenzielles Verhalten „.“ eingeführt wurde. Sie beheben also den Mangel des jeweiligen Modells, das entweder nur paralleles Verhalten oder nur sequenzielles Verhalten (Rekursion) darstellen kann.

Ein *Process Rewrite System* $\Pi = (Q, \Sigma, I, \rightarrow, F)$ ist wie folgt definiert: Q ist eine endliche Menge von atomaren Prozessen, Σ ist ein endliche Menge von Aktionen, $I \in Q$ ist der Startprozess, $\rightarrow \subseteq PEX(Q) \times (\Sigma \uplus \{\lambda\}) \times PEX(Q)$ ist die Menge der Überführungsregeln, $F \subseteq PEX(Q)$ ist die Menge der finalen Prozesse. Die Menge $PEX(Q)$ enthält alle Prozess-algebraischen Ausdrücke über der Menge der atomaren Prozesse Q und den Operatoren „.“ (sequenzielles Verhalten) und „||“ (paralleles Verhalten). λ ist als *keine Aktion* zu interpretieren. Alle Definitionen und Details zu Process Rewrite Systems können in [May98] gefunden werden.

Die Erstellung von Abstraktionen (als Process Rewrite Systems dargestellt) aus gegebenem Quellcode wird allgemein in [BZ08b] und für BPEL in [BZ08a] diskutiert. Da der Quellcode nur den Entwicklern eines Services zur Verfügung steht, soll zusammen mit dem Service auch dessen Abstraktion erstellt werden. Die Angebots- und Nutzungsschnittstellen müssen auch in der Abstraktion zur Verfügung stehen. Wir definieren die dafür genutzte Anpassung der Process Rewrite Systems namens *Stripped Process Rewrite Systems* $\Pi_C = (Q_C, \Sigma_C, \rightarrow_C, R_C, P_C, M_C)$ wie folgt: Q_C ist eine endliche Menge von Prozessen, Σ_C ist eine endliche Menge von Aktionen ausgehend von dieser Komponente, $\rightarrow_C \subseteq PEX(Q_C) \times \Sigma_C \times PEX(Q_C)$ ist die endliche Menge der Überführungsregeln, R_C ist eine endliche Menge von Nutzungsschnittstellen, also aufrufbare Schnittstellen (Methoden) des Services, P_C ist eine endliche Menge von Angebotsschnittstellen und $M_C : P_C \mapsto Q_C$ ist eine Abbildung von Angebotsschnittstellen auf atomare Prozesse $q \in Q_C$, q identifiziert dabei den Startpunkt der aufgerufenen Methode.

Die Abstraktion eines Services verbirgt also den Datenfluss. Darüber hinaus ist es möglich, die erstellten Regeln mit der Aktion λ weiter zur reduzieren, soweit sie den abgebildeten Kontrollfluss nicht beeinflussen. Wie gezeigt wurde, bleiben entscheidende Quellcode-Passagen, insbesondere Algorithmen und Daten verborgen. Trotzdem ist es aufgrund der Nutzung von Process Rewrite Systems möglich, das sequenzielle und parallele Verhalten (inklusive rekursiven Rückaufrufen und Referenzparametern) eines Services zu erfassen.

3 Verifikationsprozess

In diesem Kapitel werden wir den Verifikationsprozess vorstellen, wie er bereits in anderen Arbeiten [BZ08b, BZ08a] dargestellt wurde.

Die zu beantwortende Frage lautet, ob für die durch die Abstraktion Π_S der Anwendung S beschriebene Sprache $L(\Pi_S)$ gilt $L(\Pi_S) \subseteq \overline{L(P_C)}$, wobei $L(P_C)$ die reguläre Sprache ist[4], die das Protokoll P_C des Services C beschreibt. Um diese Frage zu beantworten, wurde ein automatischer Verifikationsprozess definiert (Abbildung 3).

[4] $\overline{L(P_C)}$ die komplementäre Sprache zu $L(P_C)$

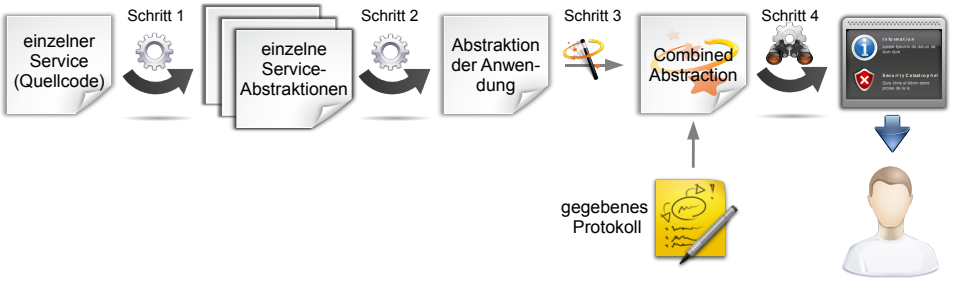

Abbildung 3: Verifikationsprozess für Anwendungen, deren einbindbare Services bekannt sind.

Dieser erstellt im ersten Schritt aus jedem Service ein Stripped Process Rewrite System. Dieses wird dem Ersteller (z. B. Softwarearchitekt) der Anwendung S zur Verfügung gestellt (z. B. integriert in die WSDL). Je nach Wunsch des Architekten werden die Anwendungen verknüpft (Schritt 2). Dabei müssen aber nicht alle Services statisch verknüpft werden. Es ist ebenso möglich, die Schnittstellen nicht fest zu binden. In diesem Fall würden die Services dynamisch zur Laufzeit gewählt. Der Verifikationsprozess wählt dann aus allen verfügbaren Services die aus, die die Nutzungsschnittstelle implementieren. So ist sichergestellt, dass alle Möglichkeiten überprüft werden.

Die entstandene Anwendungsabstraktion Π_S wird mittels einer in früheren Arbeiten [BZ08b, BZ08a] definierten Operation mit dem invertierten Protokoll geschnitten (Schritt 3). Es resultiert eine sogenannte „Combined Abstraction“. Im nachfolgenden vierten Schritt wird geprüft (Modellprüfung), ob es eine Verletzung des gerade geprüften Protokolls gibt, beziehungsweise es werden Aufrufreihenfolgen (Gegenbeispiele) dargestellt, die zu einer Verletzung des betrachteten Protokolls führen. Dies sichert die Nachvollziehbarkeit der berechneten Aussage. Für jedes zu prüfende Protokoll werden alle Schritte durchgeführt.

Da es sich um konservative Abstraktionen und Operationen handelt, ist sichergestellt, dass kein Gegenbeispiel existert, wenn bei der Modellprüfung keines gefunden wurde. Im gegensätzlichen Fall ist es aber möglich, dass ein berechnetetes Gegenbeispiel nicht im Quellcode nachvollzogen werden kann, z. B. weil die Abstraktionen nicht exakt genug waren. Diesem Problem kann mit einer CEGAR-Schleife [E. 00] begegnet werden.

Der vorgestellte Verifikationsprozess benötigt auf der Ebene der Modellprüfung keinen Zugriff auf den Quellcode. Nur die Service-Abstraktionen sind nötig. Der Blackbox-Charakter und die Eigenständigkeit der Services auch auf der Ebene der Abstraktionen bleiben ansonsten erhalten. Damit ist dieser Verifikationsprozess für einen Einsatz für auf Services basierenden Anwendungen geeignet. Diese können vor der Auslieferung verifiziert werden. Genauer, es kann eine Aussage darüber getroffen werden, welche Services gewählt werden können, ohne die definierten Eigenschaften zu verletzen. Die Services können dynamisch ausgetauscht werden, soweit die zur Auswahl stehenden Services bekannt sind.

Es wird also die Aussage geliefert, welche der Services in die Anwendung integriert werden können, ohne dass die aus der (IT-)Compliance oder den Geschäftsregeln abgeleiteten Protokolle verletzt werden. Die Integration neuer Services kann so auch gut unterstützt werden, da überprüfbar ist, ob diese die Funktionsfähigkeit der Anwendung einschränken

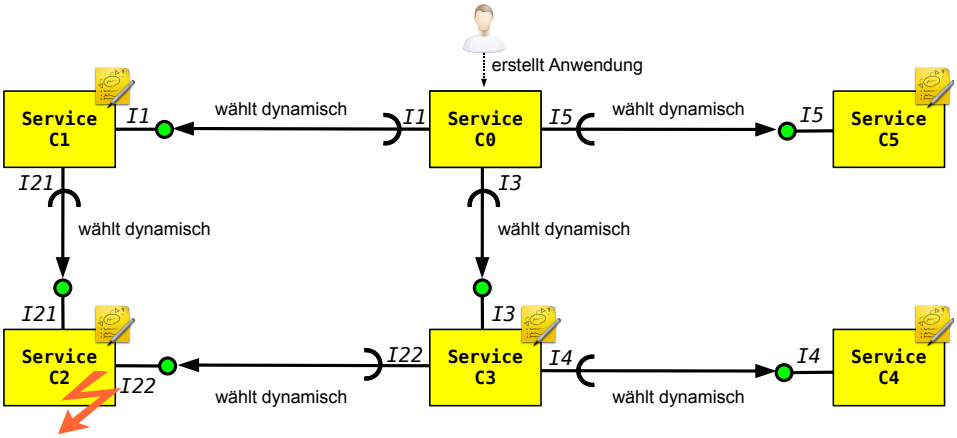

Abbildung 4: Szenario: Fehler in einer dynamisch erstellten Service-orientierte Architektur nach dem zweiten Einbinden des Services `C2`.

oder verhindern.

Dieser Verifikationsprozess ist aber noch nicht für Service-orientierte Architekturen geeignet, da in dieser die Services ihre benötigten Partner dynamisch wählen können. Gegebenenfalls wird dazu ein Service-Verzeichnis angefragt. Dies kann bewirken, dass eine Kette von Services gebildet wird. Diese Services sind erst zur Laufzeit bekannt. Der Ersteller der Anwendung hat nur die Interfaces definiert. In Abbildung 4 wird ein solches Szenario dargestellt. Der zuletzt gewählte (gebundene) Service (im Beispiel `C2`) löst einen Fehler aus und führt die dynamisch erstellte Anwendung damit (potenziell) zum Absturz. Ein solches Problem kann auch durch rekursive Rückaufrufe ausgelöst werden. Aus unserer Sicht sind Verletzungen der Geschäftsregeln oder Compliance gravierender, da der Fehler möglicherweise nicht entdeckt wird (da die Anwendung nicht zwangsläufig abstürzt).

4 Integration in Service-orientierte Architekturen

Der im vorhergehenden Kapitel vorgestellte Verifikationsprozess kann nur mangelhaft mit Services umgehen, wenn diese dynamisch zur Laufzeit gewählt werden und zum Zeitpunkt der Erstellung der Anwendung nur das Interface bekannt war. Diese Situation finden wir häufig in Service-orientierten Architekturen vor.

Um den Charakter der dynamischen Auswahl von Services zu erhalten, sind verschiedene Änderungen nötig, die wir im Folgenden erläutern werden.

Als wichtigste Änderung kann die Überprüfung der Anwendung nicht mehr zur Erstellungszeit durchgeführt werden. Zu diesem Zeitpunkt steht im Zweifelsfall kein verwendeter Service zur Verfügung. Die Verifikation kann also erst zum Zeitpunkt der Bindung eines Services durchgeführt werden.

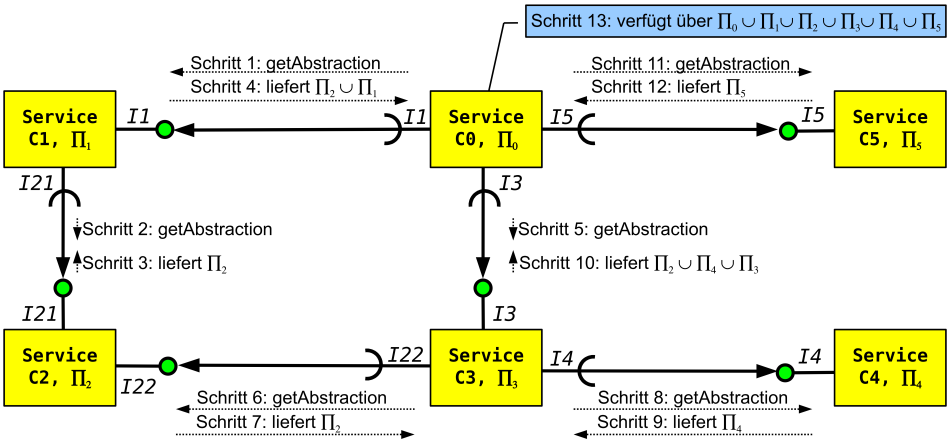

Abbildung 5: Phase 1: Erstellen einer Anwendungsabstraktion bei dynamischer Komponentenauswahl.

Um dies zu ermöglichen, muss es möglich sein, die Abstraktion eines Services vor dem Binden abzufragen. Aus diesem Grund schlagen wir vor, in das Metamodell der Services eine Methode `getAbstraction` zu integrieren, die das Stripped Process Rewrite System des Services liefert. Da dieser Service aber ebenso andere Services dynamisch einbinden kann, die ein Problem in der Gesamtanwendung auslösen können, ist es nötig, dass vorher die Abstraktionen von diesen eingebundenen Services abgefragt wird. Deshalb bindet die Methode `getAbstraction` bereits die Abstraktionen der benutzten Komponenten ein[5] und vereint diese zu einem Stripped Process Rewrite System. Dies entspricht dem zweiten Schritt im Verifikationsprozess. Es bleibt nur die Besonderheit, dass kein Process Rewrite System erstellt wird, sondern ein Stripped Process Rewrite System, also nicht alle Nutzungsschnittstellen bereits gebundenen wurden. Dieses rekursive Vorgehen ist in Abbildung 5 an einem abstrakten Beispiel dargestellt. Im Beispiel werden die Aufrufe von `getAbstraction` der Übersichtlichkeit halber nacheinander ausgeführt. Von einer parallelen Ausführung ist aber auszugehen.

Am Ende dieser ersten Phase steht der Anwendung eine vollständige Anwendungsabstraktion zur Verfügung (in Abbildung 5 nach dem 12. Schritt). Da zu diesem Zeitpunkt noch nicht überprüft wurde, ob die Services in die Anwendung integriert werden dürfen, wurden diese noch nicht gebunden. Zu diesem Zeitpunkt befinden sie sich in einem Zustand „Verbindungsanfrage“.

Nach dieser Phase beginnt die Verifikationsphase. In dieser werden die „eingesammelten“ Stripped Process Rewrite Systems der verschiedenen benutzten Services an jeden eingebundenen Service weitergegeben, so dass dieser bestimmen kann, ob die Anwendung das eigene Protokoll einhält. Dafür wird die Methode `isValid` verwendet. Über diese wird das Stripped Process Rewrite System der anderen Services der Anwendung weitergege-

[5] Es wird davon ausgegangen, dass jederzeit bestimmbar bleibt, welchem Service eine Überführungsregel (des Process Rewrite Systems) zugeordnet ist.

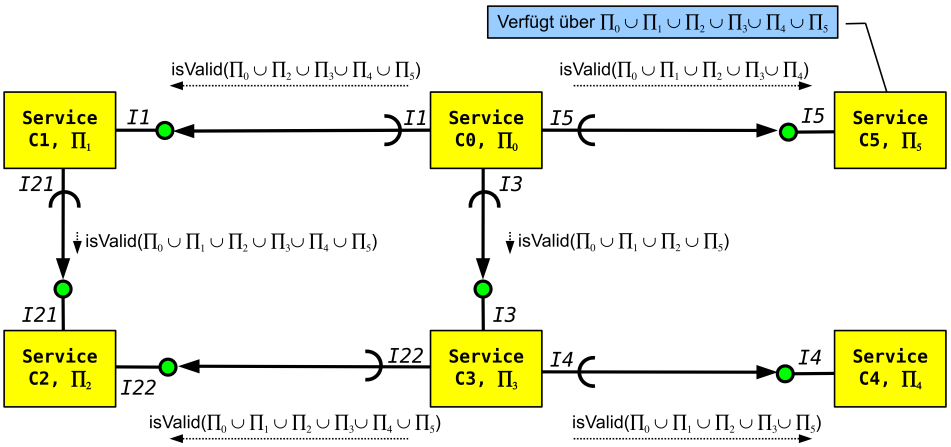

Abbildung 6: Phase 2: Verifikationsphase, Überprüfung aller Protokolle.

```
interface MetaServiceModel{
   SPRS getAbstraction();
   Status isValid(SPRS);
}
```

Der Datentyp `SPRS` stellt dabei die Stripped Process Rewrite Systems dar, während der Aufzählungstyp `Status` die Rückgabe des Ergebnisses der Modellprüfung erlaubt.

Abbildung 7: Interface für Services einer Service-orientierte Architektur.

ben und der Auftrag erteilt, die übergebene Abstraktion in die zur Verfügung stehenden Abstraktionen einzubinden. Anschließend kann jeder Service überprüfen, ob das eigene Protokoll in der Gesamtanwendung verletzt werden würde und das Ergebnis dem Auftraggeber zurückmelden. Der Ablauf in dieser Phase ist in Abbildung 6 dargestellt.

Am Ende dieser zweiten Phase verfügt die implementierte Anwendung über die Information, ob sie ein Verhalten definiert, dass ein Protokoll verletzt, oder alle Regeln einhält. Sollte keiner der eingebundenen Services ein Problem melden, kann die Anwendung automatisch ausgeliefert werden. Andernfalls muss das nun benannte Problem (Gegenbeispiel) aufgelöst werden.

Anmerkung 2 *Es wird also eine Erweiterung des Metamodells der verwendeten Services benötigt. Diese Erweiterung wird hier als Interface dargestellt, welches jeder Service implementieren muss. Die Definition dieses Meta-Interfaces* `MetaServiceModel` *ist in Abbildung 7 dargestellt. Eine solche Anpassung kann auch im Verzeichnisdienst (z. B. UDDI) vorgenommen werden.*

Mit diesen Anpassungen ist es nun auch möglich, Service-orientierte Architekturen zu verifizieren, deren Bausteine dynamisch über ein Service-Verzeichnis gewählt werden. Die dynamische Erstellung einer Anwendung wird dadurch verlangsamt, da bei der Auswahl

Abbildung 8: Verifikationsprozess für Service-orientierte Architekturen.

einer neuen Komponente von allen eingebundenen Komponenten die Abstraktionen bestimmt werden müssen, um die aktuelle Systemabstraktion erstellen zu können. Für diese müssen dann die Protokolle aller Services der Anwendung geprüft werden. Erst nach Überprüfung der Services kann also die Anwendung als fertig gestellt betrachtet und benutzt werden. Der gesamte Prozess ist in Abbildung 8 dargestellt.

Das Verfahren hat also Nachteile, wenn dynamisch ständig neue Implementierungen eingebunden werden, die zeitkritisch zur Verfügung stehen sollen. Wie in einer anderen Arbeit [BZ09] ausgeführt, handelt es sich um ein sehr schweres Modellprüfungsproblem (EXPSPACE hard), welches im Zweifelsfall praktisch nicht lösbar ist.[6]

Darüber hinaus kann es ein Problem sein, dass jeder Service selber prüft, ob das eigene Protokoll in der Gesamtanwendung verletzt werden würde. Da nicht davon auszugehen ist, dass jeder Service über die Ressourcen verfügt, den benötigten Modellprüfer zu integrieren. Eine Auslagerung in einen Modellprüfungs-Service ist deshalb ratsam.

5 Verwandte Arbeiten

Viele Arbeiten zur Protokollprüfung betrachten nur lokale Protokollprüfung unter Benutzung von endlichen Automaten. Die Idee geht zurück auf Nierstrasz [Nie95], der statische Typüberprüfungen in Objekt-orientierte Anwendungen mit endlichen Automaten modellierte. Endliche Automaten besitzen den Nachteil, dass mit ihnen weder Rekursion noch paralleles Verhalten abgebildet werden kann.

In einem Vorläufer dieser Arbeit [ZS06] wird Rekursion erfasst, da Kellerautomaten genutzt werden, um die Abstraktionen darzustellen. Ebenso ist es bereits möglich, rekursive Rückaufrufe darzustellen.

Ein Ansatz für die Protokollprüfung von nebenläufigen Komponentensystemen wurde von Schmidt und anderen [HBPR02] vorgeschlagen. Dieser Ansatz kann aber keine rekursiven Rückaufrufe erfassen.

In [PP08] wird die Konformität von Verhaltensprotokollen (behavioral protocol conformance) betrachtet. Dieses Problem ist vergleichbar mit dem hier diskutierten. Das Verhaltensprotokoll erfasst aber sowohl die Abstraktion, als auch das erlaubte Verhalten. Wie in den vorhergehenden Arbeiten wird aber kein Ansatz präsentiert, wie mit dynamisch, zur Erstellungszeit unbekannten Services umgegangen wird.

Andere Arbeiten nutzen Prozessalgebren zur Darstellung des Verhaltens, z. B. CSP in

[6] Im Rahmen der referenzierten Arbeit wurde eine Obergrenze von 30 Minuten für die Modellprüfung angesetzt.

[AG97], FSP in [FUMK06] oder CCS in [SBS04]. Die beiden letztgenannten Arbeiten diskutieren aber (im Gegensatz zu [AG97]) nicht die Protokollkonformität. Dynamisch gewählte Services werden nicht betrachtet.

Der hier vorgestellte Ansatz passt den Ansatz aus [BZ08a] für den Einsatz in Service-orientierten Architekturen an, so dass eine Überprüfung der Protokolle auch dann möglich ist, wenn zur Erstellungszeit keine Informationen über die wählbaren Services verfügbar sind.

Andere Arbeiten erkannten bereits den Mangel an semantischen Informationen über Services und schlugen Erweiterungen vor, um diesem Mangel zu begegnen. In [TPC+03] werden Testskripte zentral abgelegt, die von allen Services, die diesen Service einbinden wollen, erfolgreich absolviert werden müssen.

Viele Arbeiten gehen aber davon aus, dass der Quellcode aller Services vorliegt oder jeder Service der zentralen Anwendung bekannt ist [FBS04].

Darüber hinaus enthalten die Protokolle in anderen Arbeiten [LPT08, FGL+08, PP08] auch das Verhalten des Services. Dieses verringert aus unserer Sicht die Austauschbarkeit der Implementierung und die Anwendbarkeit, da vom Service-Entwickler nicht nur die einzuhaltenden Regeln, sondern auch das Verhalten der Abstraktionen definiert werden muss. Um diesem Problem zu begegnen, unterscheiden wir zwischen Abstraktionen und Protokollen und erzeugen die Abstraktionen automatisch mittels Übersetzerbautechniken. Dies hat darüber hinaus den Vorteil, dass der Entwickler nur bei genau den Services von Hand Definitionen vornehmen muss, deren Ausführung durch ein Protokoll abgesichert werden muss.

6 Zusammenfassung und zukünftige Arbeiten

In dieser Arbeit wurde ein Ansatz vorgestellt, der das Blackbox-Konzept von Komponenten in Service-orientierten Architekturen so erweitert, dass Aussagen über funktionale Eigenschaften ermöglicht werden. Dies gelingt nun auch, wenn Informationen über Services erst zur Laufzeit zur Verfügung stehen. Trotz der Bekanntgabe eines Teils des Verhaltens ist das Geschäftsgeheimnis der Services weiterhin abgesichert. Dies erlaubt die Anwendung in z. B. B2B-Anwendungen.

In dem vorgestellten Verfahren wird die Funktionalität unter dem Gesichtspunkt der einzuhaltenden Interaktionsreihenfolgen betrachtet. Dies sichert nicht die vollständige Funktionalität, aber die Einhaltung von für Services vordefinierten Prozessen. Selbst wenn eine Protokollverletzung keinen Absturz eines Services zur Folge hat, ist es doch möglich, dass die Geschäftsregeln oder die (IT-)Compliance verletzt werden. Dies kann schwerwiegende Folgen haben.

Da es sich um eine Modellprüfung handelt, können Fehler ausgeschlossen werden (soweit der Quellcode konservativ erfasst wurde). Die Automatisierung unseres Ansatzes wird die Entwicklungszeiten verkürzen. Ebenso können die Integrationskosten von Prozessen, an denen viele unbekannte Services beteiligt sind, verringert werden. Der Entwickler wird

von vielen „Fleißarbeiten“ entlastet. Bei der dynamischen Integration von Services ist es dann nur nötig zu überprüfen, ob die Protokolle der einzelnen Services eingehalten werden. Zur Laufzeit kann also entschieden werden, ob ein Service eingebunden werden darf oder nicht. Um dies zu tun, werden nur die Abstraktionen des Kontrollflusses der Komponenten benötigt.

Da der gesamte Verifikationsprozess vollautomatisch funktioniert, ist es für einen Softwarearchitekt oder Berater leichter möglich, einen Geschäftsprozess zu entwickeln beziehungsweise sicherzustellen, dass alle Services in diesem funktionieren.

Darüber hinaus schlagen wir eine Integration der Service-Abstraktion und des Service-Protokolls in die Schnittstellenbeschreibung (z. B. WSDL) vor. Dies sichert den transparenten Zugriff auf die Bedingungen, die bei der Integration zwingend erfüllt werden müssen. Da der Verifikationsprozess vollautomatisch abläuft, ist dessen Integration in den Kompositionsprozess leicht möglich und wie (in anderen Arbeiten gezeigt wurde) auch im zeitlichen Rahmen machbar.

Schlussendlich schlagen wir die Aufnahme von funktionalen und Maschinen verarbeitbaren Eigenschaften in das Service-Level-Agreement vor, um sicher zu stellen, dass auch die langfristige Benutzung von Komponenten beziehungsweise deren Komposition zu größeren Systemen abgesichert werden kann.

Aufgabe zukünftiger Forschung wird es sein, zu evaluieren, welche Informationen bei der dynamischen Bindung wiederverwendet werden können. Dies soll den Verifikationsprozess für dynamische Anwendungen beschleunigen.

Darüber hinaus sollte untersucht werden, welche Lösungsstrategien für einen Konflikt gefunden werden können. Dies würde zu sich selbst verändernder Software führen, die die eingebundenen Services sowohl nach funktionalen, als auch nach nicht-funktionalen Gesichtspunkten (z. B. Kosten) auswählt.

Literatur

[AG97] R. Allen und S. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[BZ08a] Andreas Both und Wolf Zimmermann. Automatic Protocol Conformance Checking of Recursive and Parallel BPEL Systems. In *IEEE Sixth European Conference on Web Services (ECOWS '08)*, Seiten 81–91, November 2008.

[BZ08b] Andreas Both und Wolf Zimmermann. Automatic Protocol Conformance Checking of Recursive and Parallel Component-Based Systems. In Michel R. V. Chaudron, Clemens A. Szyperski und Ralf Reussner, Hrsg., *Component-Based Software Engineering, 11th International Symposium (CBSE 2008)*, Jgg. 5282 of *Lecture Notes in Computer Science*, Seiten 163–179. Springer, October 2008.

[BZ09] Andreas Both und Wolf Zimmermann. A Step Towards a More Practical Protocol Conformance Checking Algorithm (wird erscheinen). In *35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA2009)*. IEEE Computer Society, August 2009.

[E. 00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. *Counterexample-guided abstraction refinement*. In Proccedings of CAV, Lecture Notes in Computer Science (LNCS) 1855, Seiten 154–169, Springer-Verlag, 2000.

[FBS04] Xiang Fu, Tevfik Bultan und Jianwen Su. Analysis of interacting BPEL web services. In *WWW'04: Proc. of the 13th int. conf. on World Wide Web*, Seiten 621–630, New York, USA, 2004. ACM.

[FGL+08] Alessandro Fantechi, Stefania Gnesi, Alessandro Lapadula, Franco Mazzanti, Rosario Pugliese und Francesco Tiezzi. A Model Checking Approach for Verifying COWS Specifications. In José Luiz Fiadeiro und Paola Inverardi, Hrsg., *FASE*, Jgg. 4961 of *Lecture Notes in Computer Science*, Seiten 230–245. Springer, 2008.

[FLNT98] J. Freudig, W. Löwe, R. Neumann und M. Trapp. Subtyping of context-free classes. In *Proc. 3rd White Object Oriented Nights*, 1998.

[FUMK06] Howard Foster, Sebastián Uchitel, Jeff Magee und Jeff Kramer. Model-Based Analysis of Obligations in Web Service Choreography. In *AICT/ICIW*, Seite 149, 2006.

[HBPR02] H. W. Schmidt, B. J. Krämer, I. Poernemo und R. Reussner. Predictable component architectures using dependent finite state machines. In *Proc. of the NATO Workshop Radical Innovations of Software and Systems Engineering in the Future*, Jgg. 2941 of *Lecture Notes in Computer Science*, Seiten 310–324. Springer, 2002.

[LPT08] Alessandro Lapadula, Rosario Pugliese und Francesco Tiezzi. Service Discovery and Negotiation With COWS. *Electron. Notes Theor. Comput. Sci.*, 200(3):133–154, 2008.

[May98] Richard Mayr. *Decidability and complexity of model checking problems for infinite-state systems*. Dissertation, 1998.

[Nie95] Oscar Nierstrasz. Regular Types for Active Objects. In Oscar Nierstrasz und Dennis Tsichritzis, Hrsg., *Object-Oriented Software Composition*, Seiten 99–121. Prentice-Hall, 1995.

[PP08] Pavel Parizek und Frantisek Plasil. Modeling of Component Environment in Presence of Callbacks and Autonomous Activities. In Richard F. Paige und Bertrand Meyer, Hrsg., *TOOLS (46)*, Jgg. 11 of *Lecture Notes in Business Information Processing*, Seiten 2–21. Springer, 2008.

[Reu02] R. H. Reussner. Counter-constraint finite state machines: A new model for resource-bounded component protocols. In *Proc. of the 29th Annual Conf. in Current Trends in Theory and Practice of Informatics*, Jgg. 2540 of *Lecture Notes in Computer Science*, Seiten 20–40. Springer, 2002.

[SBS04] Gwen Salaün, Lucas Bordeaux und Marco Schaerf. Describing and Reasoning on Web Services using Process Algebra. *International Conference on Web Services*, 0:43, 2004.

[TPC+03] W. T. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi und B. Xiao. Verification of Web Services Using an Enhanced UDDI Server. *Object-Oriented Real-Time Dependable Systems, IEEE International Workshop on*, 0:131, 2003.

[ZS06] Wolf Zimmermann und Michael Schaarschmidt. Automatic Checking of Component Protocols in Component-Based Systems. In Welf Löwe und Mario Südholt, Hrsg., *Software Composition*, Jgg. 4089 of *LNCS*, Seiten 1–17. Springer, 2006.