Deadlockanalyse Service-orientierter Softwaresysteme

Dissertation
zur Erlangung des
Doktorgrades der Naturwissenschaften (Dr. rer. nat.)

der

Naturwissenschaftlichen Fakultät III,
Agrar- und Ernährungswissenschaften,
Geowissenschaften und Informatik

der Martin-Luther-Universität Halle-Wittenberg

vorgelegt von

Frau Weißbach, Mandy
Geb. am 13.03.1983 in Querfurt

Gutachter:
Prof. Dr. Wolf Zimmermann
Prof. Dr. Welf Löwe

Verteidigungsdatum:
12.12.2019

Weißbach, Mandy
Deadlockanalyse Service-orientierter Softwaresysteme

Dissertation, Martin-Luther-Universität Halle-Wittenberg,
September 2019.

Zusammenfassung

In der vorliegenden Arbeit liegt der Fokus auf der Deadlockanalyse von Service-orientierten Softwaresystemen. Dabei sollen diese Softwaresysteme unbeschränkte Nebenläufigkeit, unbeschränkte Rekursion und Synchronisation enthalten dürfen. Es soll ein abstraktions-basierter Ansatz präsentiert werden, der bereits implementierte Softwaresysteme auf Deadlockfreiheit überprüfen kann. Dabei soll das Blackbox-Paradigma von Diensten in Service-orientierten Softwaresystemen unterstützt werden.

Der aktuelle Forschungsstand im Bereich Deadlockanalyse macht es weder möglich den spezifischen Anforderungen, wie das Blackbox-Paradigma von Service-orientierten Systemen gerecht zu werden, noch können Service-orientierte Softwaresysteme mit unbeschränkter Nebenläufigkeit, unbeschränkter Rekursion und Synchronisation auf Deadlockfreiheit überprüft werden.

Daher untersuchen wir in dieser Arbeit bereits bekannte Abstraktionsmethoden hinsichtlich der spezifischen Anforderungen von Service-orientierten Softwaresystemen. Es wird der aktuelle Forschungsstand betrachtet und es werden unterschiedlichen Ansätze im Bereich der Deadlockanalyse untersucht. Dabei werden die Mächtigkeit der Modelle dieser unterschiedlichen Ansätze untersucht, wie zum Beispiel Petri-Netze, im Besonderen P/T-Netze.

Es wird gezeigt, dass Petri-Netz-basierte Ansätze zu falsch positiven Ergebnissen führen können. Wobei falsch positive Ergebnisse bedeuten, dass das Service-orientierte Softwaresystem fälschlicherweise als deadlockfrei klassifiziert wird, während bei der Ausführung des Systems ein Deadlock auftreten kann. Modelle, wie zum Beispiel Petri-Netze oder Prozessersetzungssysteme werden diskutiert und Abstraktionsverfahren vorgeschlagen, um das Ziel einen konservativen Ansatz zur Deadlockanalyse für Service-orientierte Systeme mit unbeschränkter Rekursion, unbeschränkter Nebenläufigkeit und Synchronisation zu ermöglichen und falsch positive Ergebnisse zu vermeiden.

Das vorgeschlagene Verfahren kann die spezifischen Anforderungen von Service-orientierten Softwaresystemen gerecht werden und falsch positive Ergebnisse ausschließen.

Schlüsselwörter: Petri-Netz, P/T-Netz, Process Rewrite System, Modellierung, Abstraktion, abstraktions-basiert, Deadlockanalyse, unbeschränkte Rekursion, unbeschränkte Nebenläufigkeit, Synchronisation, Falsch Positive, Falsch Negative

Abstract

In the present PhD thesis, the focus is on the deadlock analysis of service-oriented software systems. These software systems should be able to contain unbound recursion, unbound concurrency and synchronization. An abstraction-based approach, i.e. based on the source code, will be proposed, which can check already implemented software systems for deadlock freeedom. The aim is to support the black box view paradigm of services in service-oriented software systems.

The current state of the art in deadlock analysis neither makes it possible to meet the specific requirements of the black box paradigm of service-oriented software systems, nor can service-oriented software systems with unlimited concurrency, unbounded recursion and synchronization be checked for deadlock freedom.

Therefore, in this work, we are investigating well-known abstraction methods with regard to the specific requirements of service-oriented systems. We consider the current state of research and different approaches in the field of deadlock analysis, and we discuss the power of the models of these different approaches, such as Petri nets, in particular P/T-nets. They are discussed with regard to deadlock analysis of service-oriented software systems.

We show that Petri-net-based approaches can lead to false positives. Where false positives results mean that the service-oriented system has been incorrectly classified as deadlock-free, while the execution of the service-oriented system may lead to a deadlock. Models such as Petri nets or Process Rewrite Systems are discussed with the aim of conserving a conservative verification process for service-oriented systems with unbound recursion, unbound concurrency and synchronization.

With the proposed verification process, the specific requirements of service-oriented systems can be met and false positive results can be excluded.

Keywords: petri net, P/T-net, process rewrite system, modeling, abstraction-based, deadlock analysis, unbound recursion, unbound concurrency, synchronization, false positives, false negatives

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1. Einleitung

1.1. Motivation und Problemstellung

Softwaresysteme, die unser tägliches Leben vereinfachen sollen, halten Einzug in unser aller Leben. Ein Beispiel dafür sind die Anwendungen oder Apps auf unseren Smartphones. Sie stellen bestimmte Dienste zur Verfügung. Wir können den aktuellen Kontostand abfragen, die Anzahl der gelaufenen Schritte pro Tag aufzeichnen und unseren Fitnesszustand berechnen lassen, unsere E-Mails und Termine verwalten, dir wir bei Bedarf mit anderen teilen können, oder wir können uns anstehende Zugverspätungen unserer Reiseverbindung mitteilen lassen. Das sind nur einige der zahlreichen Anwendungen, die dem Nutzer von Smartphones zur Verfügung stehen. In der heutigen Zeit vergeht kein Tag an dem wir nicht mindestens eine Anwendung auf dem Smartphone nutzen [9]. Wobei es alleine im Jahr 2018 etwa 2,6 Milliarden Smartphone-Nutzer gab [20] und es wird prognostiziert, dass diese Anzahl bis 2021 auf circa 3,08 Milliarden ansteigen wird.

Im Zusammenhang mit Smartphones und der intensiven Nutzung von Apps steht auch das rasante Wachstum im Bereich *Internet der Dinge* (engl. Internet of Things, Abk. IoT). Bereits 1999 wurde der Begriff geprägt durch [5]. Aber erst mit der Entwicklung des Internets, der ständigen Erreichbarkeit der Menschen im Internet, wird das Internet der Dinge erst richtig nutzbar. Dinge aus dem alltäglichen Leben, wie zum Beispiel Kühlschränke oder Staubsauger, werden zu intelligenten Objekten. Mit Hilfe von Kommunikations- und Informationstechnik, angebunden an das Internet mit seinen mächtigen Diensten können sie über Sensoren ihren Kontext wahrnehmen, sich mit anderen Diensten vernetzen und eben mit Menschen interagieren [37]. Alles auf Basis von Diensten und deren Kommunikation.

Diese Anwendungen sind ein Beispiel für Service-orientierte Softwaresysteme. Die vom Nutzer gewünschte Funktionalität, wie zum Beispiel den Kontostand abfragen, wird ihm über einen Dienst, der auf einen entfernten Server zur Verfügung gestellt wird, bereitgestellt. Dabei werden die Daten vom Nutzer über die App ein- und weitergegeben. Die Daten werden vom jeweiligen Dienst empfangen, verarbeitet, gespeichert und die gewünschten Informationen werden wieder an den Nutzer zurückgesendet. Wir können also von einer weltweiten extensiven Nutzung Service-orientierter Systeme sprechen [32].

Der Erfolg Service-orientierter Systeme im Vergleich zu monolithischen Systemen ist begründet durch die zahlreichen Vorteile eines solchen Systems. Ein Vorteil für Smartphone-Nutzer ist zum Beispiel, dass trotz eingeschränkter Hardware, die Smartphones, Dienste mit umfangreichen Funktionalitäten zur Verfügung gestellt werden können. Außerdem können schnell, unterschiedliche Anwendungen auch zum Beispiel in Unternehmen integriert werden (austauschbar und erweiterbar). Zusätzlich kann bei Bedarf, auch auf die Funktionalitäten externer Dienste zugegriffen werden (verteilbar). Die Systeme sind damit erweiterbar, austauschbar und verteilbar [6].

Bei der Entwicklung von Service-orientierten Systemen verfügen Architekten über eine Vielzahl verfügbarer Dienste mit unterschiedlichen oder gleichen Funktionalitäten. Dienste werden anhand der angeforderten Funktionalität ausgewählt. Fehlt ein Dienst mit einer bestimmten Funktionalität, können auch Dienste von Drittanbietern genutzt werden. Der Quellcode wird dabei normalerweise nicht mit dem Dienst ausgeliefert und ist daher nicht verfügbar.

Wenn während der Ausführung des Service-orientierten Softwaresystems Fehler auftreten, wird es meist schwierig ohne Quellcode, die Ursache der Fehler zu ermitteln. Das Verbergen des Quellcodes wird als Geheimnisprinzip oder aber auch Black-Box-Paradigma bezeichnet. Im Bereich der Service-orientierten Architekturen wird dieses Paradigma über Web Services umgesetzt.

In dieser Arbeit werden Analyseverfahren zur Vermeidung von Fehlern in Service-orientierten Softwaresysteme untersucht. Dabei liegt der Hauptaugenmerk auf das Auftreten möglicher Deadlocks. Deadlocks sind Programmverklemmungen, die in Folge von Synchronisationen nebenläufiger Prozesse entstehen können und zählen zu den sogenannten *Heisenbugs*. *Heisenbugs* sind Fehler, die nur gelegentlich auftreten und daher extrem schwer zu finden sind, weil sie nur selten reproduzierbar und damit nur selten behebbar sind [25]. Das Problem der *Heisenbugs* ist zwar schon seit der Existenz nebenläufiger Programme bekannt, trotzdem scheint es weiterhin aktuell zu sein. So enthielten nach [41] 6500 von 198000 Bug-Reports der Sun Java das Stichwort Deadlock und betrug damit einen Anteil von circa 3%. Während es im aktuellen Jahr 2019 immer noch 957 Bug Reports von 244111 *BugReports* gibt, die das Stichwort Bug Report enthalten.

Die Deadlockanalyse von Service-orientierten Systemen stellt im Vergleich zur Deadlockanalyse monolithischer Software zusätzliche Anforderungen an das Analyseverfahren:

- durch das Geheimnisprinzip steht zum Teil der Quellcode der Komponenten oder Dienste nicht zur Verfügung,
- asynchrone Aufrufe zwischen den Services führen zur Nebenläufigkeit,
- was wiederum, im Gegensatz zu monolithischen Systemen zur Folge hat, dass Dienste auf unterschiedlichen Maschinen laufen (s.o. Smartphones, Server der Dienstanbieter) können.

Interaktionen wie synchrone und asynchrone Aufrufe sowie rekursive Rückrufe zwischen den Diensten müssen modelliert und analysiert werden. Aktuelle Entwicklungen gehen auf die Verwendung von sogenannten Microservices hin. Einzelne Funktionalitäten werden bis auf eine feine Granularitätsstufe heruntergebrochen und in einzelne Services gekapselt. Diese Systeme sind noch besser skalierbar, da viele Instanzen erzeugt werden können, die parallel zueinander laufen können [34]. Die Interaktionen zwischen den Microservices steigen aber dadurch im Vergleich zu regulären Service-orientierten Systemen stark an. Es ist davon auszugehen, dass solche *Heisenbugs* häufiger auftreten werden. Das Testen solcher Systeme wird aber durch die starke Verteilung schwieriger [42]. Werkzeuge zur Deadlockanalyse von Service-orientierten Systemen mit parallelen und rekursiven Verhalten sind daher von besonderer Bedeutung [42].

Es existiert bereits eine große Anzahl an Arbeiten im Bereich der Deadlockanalyse [1], [21]. Eine ausführliche Betrachtung findet im Kapitel 3 statt.

[2] stellt einen verfeinerungs-basierten Ansatz zur Deadlockanalyse vor. Von der abstrakten Modellierung zum implementierten System. Dieser Ansatz ist zwar für eine erste Implementierung geeignet, es ist jedoch schwierig, die Konsistenz eines Modells über die Zeit hinweg aufrecht zu erhalten. Die Umsetzung erfordert außerdem disziplinierte Programmierer. Bereits implementierte Komponenten oder Services können mit dieser Methode unter Umständen auch analysiert werden. Jedoch muss von der Implementierung zur Petri-Netz-Darstellung abstrahiert werden.

Da Petri-Netz-basierte Ansätze kein rekursives Verhalten modellieren können, kann eine solche Abstraktion eine Implementierung mit rekursiven Verhalten nicht modellieren. So können zum Beispiel rekursive Prozeduraufrufe zwischen einzelnen Komponenten oder Services, nicht erfasst werden und es können *falsch positive* Ergebnisse auftreten [56].

Um rekursives Verhalten zu modellieren, können Kellersysteme benutzt werden. Paralleles Verhalten hingegen kann damit nicht modelliert werden [15], [16], [56].

Daher müssen Modelle betrachten werden, die paralleles und rekursives Verhalten modellieren können. Mit Mayr's Process Rewrite Systems (PRSs) kann paralleles und rekursiven Verhalten modelliert werden [29]. Paralleles und rekursives Verhalten kann so modelliert werden, dass *falsch positive* Ergebnisse bei der Protokollkonformitätsprüfung nicht mehr auftreten können.

Aus diesem Grund wird untersucht inwieweit Mayr's PRS als formales Modell zur Deadlockanalyse in Service-orientierten Softwaresystemen genutzt werden kann. Die im Folgenden aufgestellten Eigenschaften und Anforderungen an das System sollen dabei gelten:

Die in dieser Arbeit betrachteten Service-orientierten Systeme besitzen folgenden Eigenschaften:

- unbeschränkte Rekursionstiefe einschließlich rekursiver Rückrufe,
- unbeschränkte Parallelität beziehungsweise Nebenläufigkeit und
- Synchronisationsbarrieren.

Daraus ergeben sich für diese Arbeit folgende Anforderungen an den zu entwickelnden Ansatz:

- Modellierung der Rekursionstiefe ohne Beschränkung einschließlich rekursiver Rückrufe ohne Beschränkung
- Modellierung unbeschränkter Parallelität beziehungsweise Nebenläufigkeit,
- Modellierung eines Barriere-abhängigen Synchronisationsmechanismus,
- Vorstellung eines abstraktions-basierten Ansatz, der es prinzipiell ermöglicht, bereits implementierte Dienste zu untersuchen und
- die Erhaltung des Black-Box-Paradigma für Service-orientierte Systeme.

1.2. Wissenschaftliche Fragestellung

In dieser Arbeit sollen bisherige zugrunde liegende Verfahren zur Deadlockanalyse vorgestellt und untersucht werden, inwieweit diese im Bereich der Service-orientierten Systeme eingesetzt werden können. Dabei sollen die Service-orientierten Systeme unter den oben genannten Anforderungen und Eigenschaften nicht zu falsch positiven Ergebnissen führen.

In dieser Arbeit sollen daher folgende Fragestellungen beantwortet werden:

1. Führen die bisherigen Ansätze zur Deadlockanalyse hinsichtlich den aufgestellten Anforderungen zu falsch positiven Ergebnissen?
2. Wenn bisherige Lösungen zu falsch positiven Ergebnissen führen, können die diese Ansätze zur Deadlockanalyse erweitert werden oder gibt es andere formale Modelle, um die aufgestellten Anforderungen und Eigenschaften zu erfüllen?
3. Ist die Umsetzung des vorgeschlagenen Ansatzes zur Deadlockanalyse prinzipiell möglich?

1.3. Lösungsansatz der Arbeit

Um die Fragen beantworten zu können, wird die Arbeit in zwei Etappen unterteilt:

Deadlockanalyse

Als Grundlage wird die Beispielsprache $\mathcal{XYZ}$ definiert, deren sequentielles und nebenläufiges Verhalten mit einer operationalen Semantik beschrieben wird. Die Basis unserer operationalen Semantik bilden die sogenannten Kaktuskeller. Der Ursprung der Kaktuskeller liegt in der Arbeit von Dahl und Nygaard aus dem Jahr 1966 [19]. Kaktuskeller oder Kaktuskeller ermöglichen die Modellierung von sequentiellen und nebenläufigen Verhalten. Aus diesem Grund heißt die in dieser Arbeit für die Beispielsprache $\mathcal{XYZ}$ definierte Semantik auch Kaktuskeller-Semantik.

Sequentielles und nebenläufiges Verhalten

Um Untersuchungen von Verfahren zur Erkennung von Deadlockfreiheit zu ermöglichen, wird die entworfene operationale Semantik aus Etappe 1 genutzt (Kap. 4). Anhand eines Beispiels wird untersucht, ob bereits existierende Petri-Netz-basierte Ansätze, wie die *Workflow Netze* von van der Aalst's [2], unter bestimmten Voraussetzungen zu falschen Aussagen über die Abwesenheit von Deadlocks führen können. Schließlich wird eine Abstraktion nach dem Verfahren von [29] erstellt, die auf Mayr's Prozessersetzungssystemen [40] basiert, angegeben. Diese Verfahren lassen korrekte Aussagen mit kleinen Einschränkungen zur Abwesenheit von Deadlocks zu. Danach werden Varianten zwischen den Extremen gesucht. Mit den gewonnenen Erkenntnissen soll eine geeignete Methode zur Deadlockanalyse von Service-orientierten Softwaresystemen unter Einhaltung der in dieser Arbeit definierten Bedingungen an Service-orientierte Softwaresysteme (Kapitel 6) vorgeschlagen werden können.

1.4. Aufbau der Arbeit

Die vorliegende Arbeit ist folgendermaßen aufgebaut:

In Kapitel 2 werden die verwandten Arbeiten rund um das Thema *Deadlockanalyse in Service-orientierten Softwaresystemen* vorgestellt.

Im darauffolgenden Kapitel 3 werden die Grundlagen erarbeitet. Es werden Service-orientierte Softwaresysteme definiert und Ansätze zur formalen Modellierung von sequentiellen und nebenläufigen Softwaresystemen vorgestellt. Dabei werden die Prozessersetzungssysteme (engl. Process Rewrite Systems) nach Mayr [39], sowie auch Petri-Netze [44] nach Carl A. Petri eingeführt.

Mittelpunkt der Arbeit stellen die Kapitel 4, 5, 6 und 7 dar. Hier werden die Hauptresultate der Arbeit vorgestellt. Dazu wird in Kapitel 4 eine operationale Kaktuskeller-Semantik eingeführt, um das Ausführungsverhalten der Beispielsprache $\mathcal{XYZ}$ zu beschreiben. Die Syntax der Beispielsprache ist im Anhang A zu finden. Im Kapitel 5 werden Grenzen bisheriger Petri-Netz-basierter Lösungsansätze unter Einbeziehung der in dieser Arbeit aufgestellten Randbedingungen von Service-orientierten Softwaresystemen aufgezeigt. Im Anschluss, im Kapitel 6 wird ein neuer Ansatz zur Deadlockanalyse vorgestellt. Und im letzten Kapitel des Hauptteils, Kapitel 7, wird untersucht, inwieweit der Kompositionsmechanismus entscheidend für die Wahl des zugrunde liegende Abstraktionsmodells ist.

Abschließend werden in der Zusammenfassung und im Ausblick in Kapitel 8 auf einzelne Gesichtspunkte eingegangen, das Erreichte zusammengefasst und mögliche Ansatzpunkte für weitere Verbesserungen und neue Fragestellungen vorgestellt.

2. Verwandte Arbeiten

In dieser Arbeit sollen Modelle beziehungsweise vorhandene Analyseverfahren als Grundlage zur Deadlockanalyse von Service-orientierten Systemen untersucht werden. Um einen Ausgangspunkt für die Einschätzung und Wertung der verwandten Arbeiten zu ermöglichen, werden im Folgenden Klassifikationskriterien aufgestellt.

2.1. Klassifikationskriterien für Verwandten Arbeiten

In der Einleitung wurden Annahmen aufgestellt, die das zu untersuchende Service-orientierte System haben kann. Außerdem wurden Anforderungen an das Analyseverfahren aufgestellt, die erfüllt werden sollen. Aus diesen Annahmen und Anforderungen ergeben sich folgende Kriterien, die die Grundlage zur Klassifizierung der verwandten Arbeiten sein sollen:

- Modellierung von:
  (a) rekursiven oder sequentiellen Verhalten,
  (b) rekursiven Rückrufen (engl. recursive callbacks) und
  (c) parallelen oder nebenläufigen Verhalten.
- Enthalten die realen Systeme:
  (d) unbeschränkt asynchrone Aufrufe,
  (e) unbeschränkt synchrone Aufrufe und
  (f) barriere-abhängige Synchronisationsmechanismen.
- Deadlockanalyseverfahren:
  (g) abstraktions-basiert und

(h) anwendbar auf Service-orientierte Systeme.

Diese Kriterien werden nun kurz im Kontext der Arbeit erläutert, um eine exakte Einordnung der verwandten Arbeiten zu ermöglichen.

Rekursives Verhalten

Rekursion ist ein wichtiges Konzept in der Informatik [10]. Viele bekannte Algorithmen wie zum Beispiel der *Teil-und-Herrsche*-Algorithmus oder die Berechnung der Fakultät einer Zahl wird rekursiv implementiert. Auch in Service-orientierten Systemen kommt es zur Rekursion oder rekursiven Rückrufen über Aufrufe zwischen den Services. Aus diesem Grund, sollte bei der Systemmodellierung auch das rekursive Verhalten betrachtet werden. Um rekursives oder sequentielles Verhalten zu modellieren, können Pushdown-Automaten oder Kellerautomaten [31], [55], [35] benutzt werden [15], [16] oder Derivate wie die Kaktuskeller [28], [19] beschrieben werden. Weitere Möglichkeit sind Mayr's PRS [39], oder aber auch rekursive Petri-Netze [50].

Paralleles und nebenläufiges Verhalten

Wie die Rekursion ist auch die Parallelität und Nebenläufigkeit ein Schlüsselkonzept der Informatik. Viele Aufgaben werden in kleinere Aufgaben zerteilt, um dann parallel bearbeitet zu werden. In dieser Arbeit können Services andere Services asynchron aufrufen oder von diesen aufgerufen werden und damit unabhängig vom Aufrufer ausgeführt werden (vgl. Def. 3.9, 3.10). Die Modellierung von parallelen Verhalten basiert oftmals auf Petri-Netze [44], [47], [8].

Synchronisationsmechanismen

In diesem Ausführungsmodell existieren zwei Möglichkeiten asynchrone Prozeduren oder Funktionen zu synchronisieren. Zum einen besteht die Möglichkeit zur Synchronisation, wenn der Aufrufende eine explizite Synchronisationsanweisung (*sync*-Anweisung in dieser Arbeit) erhält, die barriere-basierte Synchronisation. Die Ausführung des Aufrufers wird dadurch blockiert und auf die Rückkehr der asynchronen Prozedur gewartet. Zum anderen wird der Aufrufer und der Aufrufende synchronisiert, wenn beide mit der Ausführung ihres Prozesses fertig sind (Erreichen der *return*-Anweisung in dieser Arbeit).

Abstraktions-basierte Verfahren

Bei den abstraktions-basierten Verfahren [11] wird das Verhalten vom Quellcode abstrahiert. Im Gegensatz zu den verfeinerungs-basierten Verfahren. Bei den verfeinerungsbasierten Verfahren (z.B. [17], [2]) wird das Verhalten eines Services spezifiziert und dann bis zur Implementierung verfeinert.

Anwendbarkeit auf Service-orientierte Systeme

Service-orientierte Systeme bestehen aus Services, deren Implementierung nicht zur Verfügung steht (Geheimnisprinzip). Aus diesem Grund, wird dieses Kriterium auch betrachtet. Ein vorgeschlagenes Verfahren zur Deadlockanalyse sollte das Geheimnisprinzip nicht verletzten [14].

Deadlockanalyse

Es existiert bereits eine große Anzahl an Arbeiten im Bereich der Deadlockanalyse [1], [21], [24], [36], [30], [47]. Viele Arbeiten [1], [47] berücksichtigen keine Rekursion oder rekursive Rückrufe, weil die Verfahren auf Petri-Netzen basieren. Petri-Netz-basierte Ansätze können kein rekursives Verhalten modellieren.

2.2. Klassifikation verwandter Arbeiten

Im Folgenden werden die verwandten Arbeiten vorgestellt und die Erfüllung der Kriterien diskutiert.

1. Petri-Netz-basierten Ansätze werden zum Beispiel in Arbeiten von [1] und [53] genutzt. In [1], [2] werden sogenannte Workflow Netze (WF-nets) vorgestellt, eine Unterklasse von Petri-Netzen mit einer Eingangsstelle und einer Ausgangsstelle. Diese Eingangs- und Ausgangsstellen dienen zur Komposition der einzelnen Netze. Die Komposition wird eher intuitiv erläutert, da eine präzise Definition fehlt. Es wird nicht klar, wie zweimalige Aufrufe bei der Modellierung dargestellt werden, ob der Aufrufkontext modelliert wird oder nicht. In diesen Arbeiten werden Rekursion und rekursive Rückrufe nicht bearbeitet.

2. *Woflan* [53] ist ein Petri-Netz-basiertes Werkzeug zur Analyse von Abläufen von Geschäftsprozessen. Auch in dieser Arbeit werden Rekursion oder rekursive Rück-

rufe nicht betrachtet. Die Verfahren, auf denen diese Arbeiten beruhen sind verfeinerungs-basierte Ansätze.

3. In [51] wird ein statisches Verfahren zur Deadlockanalyse von *C#* Programmen vorgestellt. Mithilfe von *Roslyn* [27] wird vom Quellcode zu einem auf Promela-basierten Modell abstrahiert, um anschließend mit generierten LTL Formeln die Deadlockanalyse durchzuführen. Diese Arbeit betrachtet keine unbeschränkte Parallelität oder Rekursion.

4. Das Verifikationswerkzeug *Armus* wird in [18] vorgestellt. Armus kann unterschiedliche barierre-basierte Synchronisationskonzepte behandeln. In der Arbeit wird keine Rekursion betrachtet.

5. Ein verfeinerungs-basierter Ansatz zur Deadlockanalyse für Microservices wird von Camilli et al. vorgestellt [17]. Ein System bestehend aus Microservices wird mithilfe von *CONDUCTOR* spezifiziert - ein Werkzeug zur Orchestrierung von Microservices [7]. In dieser Arbeit wird keine Rekursion betrachtet. Die *CONDUCTOR* Spezifikationen werden mit dem Tool *conductor2pn* in Petri-Netze überführt, auf denen die Deadlockanalyse durchgeführt wird.

6. Bouajjani et al. [13] schlägt einen abstraktions-basierten Ansatz vor, um die Kontrollstrukturen rekursiver paralleler Programme (z.B. Cilk, X10, Multilisp) zu modellieren. Der Ansatz basiert auf rekursiven Vektoradditionssystemen (engl. vector addition systems [33]. Sie untersuchen die Entscheidbarkeit von Erreichbarkeitsproblemen. Es scheint aber so, als ob das System Zustände erreichen kann, in denen das Verfahren, das Erreichbarkeitsproblem und damit die Deadlockanalyse unentscheidbar wird. Weder die Analyse von Services noch die Analyse von Service-orientierten Systemen wird in der Arbeit erwähnt.

7. In [50] werden rekursive und algebraische Workflow Netze (RecWF-Nets) eingesetzt, um autonome Agenten zu modellieren, die Güter von einem Standort zum anderen Standort bringen. Die hier erstmals eingeführten rekursiven Petri-Netze können dynamische Prozesse modellieren. Deadlocks können hier nur auftreten, wenn die Interaktion zwischen zwei Klienten (z.B., geteilte Attribute) einer vorher festgelegte Vorbedingung invalidieren. Aus diesem Grund werden Algorithmen vorgeschlagen, um diese Situationen und damit Deadlocks ausschließen zu können. Rekursion im Sinne dieser Arbeit wird nicht modelliert.

8. Eine weitere Arbeit, die auf Petri-Netzen basiert, ist [36]. Mertens analysiert Webservice-basierte Geschäftsprozesse und abstrahiert daher von $BPEL4WS$ zu Petri-Netzen, den sogenannten Modulen. Die Petri-Netze werden erweitert um die Stelle α und ω, die den Anfang und das Ende eines Prozesses symbolisieren. Die daraus entstandenen Module können miteinander verknüpft werden. Dazu wird ein Kompositionsprozess vorgestellt. Wie die Komposition von Modulen modelliert wird,

die mehr als einmal aufgerufen werden, ist unklar. Rekursion spielt in dieser Arbeit auch keine Rolle.

9. In [30] werden rekursive und algebraische Workflow Netze (RecWF-Nets) eingesetzt, um paralleles und auch rekursives Verhalten von Prozessen im Gesundheitssystem zu modellieren. Ein großer Nachteil des Verfahrens ist, dass keine unbeschränkte Rekursion modelliert werden kann. Rekursion im Sinne dieser Arbeit wird nicht betrachtet. Traditionelle Tools zur Verifikation können nach unserem Wissen hier nicht angewendet werden.

10. In [29] wurde gezeigt, dass mit Hilfe von Mayr's Process Rewrite Systems (PRSs) paralleles und rekursives Verhalten sowie Ausnahmebehandlungen modelliert werden können. Hier wurde im Zusammenhang mit der Protokollkonformitätsprüfung gezeigt, dass paralleles und rekursives Verhalten so modelliert werden kann, dass *falsch positive* Ergebnisse bei der Protokollkonformitätsprüfung nicht mehr auftreten können. Die Deadlockanalyse war hier nicht Teil dieser Arbeit. Die Arbeit ist ein abstraktions-basierter Ansatz.

11. Weitere Modelle zur Systemmodellierung nebenläufiger Programme, im Speziellen zur Deadlockanalyse sind die sogenannten *lam* Programme, die in [24] eingeführt werden. Die Deadlockanalyse auf *lam* Programmen ist nur für rekursiv parallele Programme entscheidbar. Arbeiten, die auf diese Arbeit aufbauen sind [23]. Hier werden weitere Einschränkungen bezüglich rekursiver Typen gemacht. Die Verfahren sind abstraktions-basiert.

12. [22] betrachtet die Deadlockanalyse von nebenläufigen Objekten und benutzt dazu *Points-to*-Analysen und *May-Happen-in-Parllel* Analysen, um Abhängigkeitsgraphen aufzubauen. Sind diese Graphen azyklisch, so existiert kein Deadlock. Der Umgang mit rekursiven Rückrufen wird hier nicht behandelt. Rekursive Objekte müssen vermieden werden. Die Anwendbarkeit dieses Verfahrens auf Serviceorientierte Systeme ist unklar beziehungsweise steht nicht im Fokus der Arbeit.

Zusammenfassung

Die Klassifikation der verwandten Arbeiten wird in Tabelle 2.1 zusammengefasst. Die Einträge umfassen dabei $++$, $+$, $-$ und $--$, wobei das Zeichen $++$ bedeutet, dass die Arbeit das entsprechende Kriterium erfüllt und das Zeichen $--$, dass die Arbeit das Kriterium nicht erfüllt. Falls aus den Arbeiten nicht hervorgeht, ob das Kriterium erfüllt wird, wird das Symbol 0 benutzt.

Zur Modellierung von Systemen stehen eine Vielzahl von Möglichkeiten zur Verfügung. Die Systemabstraktionen werden auf Basis von endlichen Zustandsmaschinen [43], [52],

Prozessalgebren [4], Petri-Netzen [2], rekursiven Petri-Netzen [26], Kellersysteme [56], [16] oder Prozessersetzungssysteme [11], [54] beschrieben. Alle diese Methoden können entweder kein unbeschränktes rekursives Verhalten modellieren beziehungsweise die Deadlockanalyse auf diesen Modelle ist dann nicht mehr entscheidbar [23]. In [56] wurde gezeigt, dass das bei der Protokollkonformitätsprüfungen von Komponenten-basierten Softwaresystemen zu *falsch positiven* Ergebnissen führen kann.

Die existierenden Deadlockanalyseverfahren und deren Anwendung auf Service-orientierte Systeme werden den aufgestellten Anforderungen, wie das Black-Box-Paradigmen, nicht gerecht. Die Arbeiten [17], [2] sind zum Beispiel verfeinerungs-basiert und können damit nicht auf bereits implementierte Services angewendet werden, bzw. eine Erweiterung von Service-orientierten Systemen mit neuen Services und einer anschließenden Deadlockanalyse auf dem System, ist, im Gegensatz zu dieser Arbeit, nicht möglich.

Auch zeigen die Arbeiten [2], [36] nicht, wie mehrmalige Aufrufe von Services in Service-orientierten Systemen bei der Modellierung behandelt werden können und ob diese eine Rolle spielen könnten.

	Arbeiten	*unbeschränkte Rekursion*	*rekursive Rückrufe*	*unbeschränkte Nebenläufigkeit*	*Synchronisation*	*Abstraktions-basierter Ansatz*	*Service-orientierte Softwaresysteme*	*Deadlockanalyse*
1.	[1], [2]	−−	−−	++	++	−−	+	++
2.	[53]	−−	−−	++	++	−−	+	++
3.	[51]	−−	−−	+	+	++	+	+
4.	[18]	−−	−−	+	++	+	0	++
5.	[17]	−−	−−	++	−	−−	+	++
6.	[12], [13]	−	−−	+	++	++	−−	+
7.	[50]	+	−	+	−	++	−−	+
8.	[36]	−−	−−	++	−−	++	++	++
9.	[30]	+	0	++	0	+	+	+
10.	[11], [10], [29]	++	++	++	+	++	++	−−
11.	[24], [23]	+	−−	++	+	+	0	++
12.	[22]	−−	−−	++	++	++	0	−−

Tabelle 2.1.: Klassifizierung der verwandten Arbeiten.

3. Grundlagen

In diesem Kapitel werden die Grundlagen dieser Arbeit beschrieben. Dabei werden im ersten Abschnitt Service und Service-orientierte Systeme definiert, sowie deren Ausführungsmodell erläutert. In Abschnitt 3.2 werden die in der Arbeit genutzten Modelle der Petri-Netze, der Prozessersetzungssysteme (engl. Process Rewrite Systems) und Kaktuskeller vorgestellt. Im dritten und letzten Abschnitt 3.3 werden Modellierungsverfahren monolithischer Programme sowie von Service-orientierten Softwaresystemen vorgestellt.

3.1. Services und Service-orientierte Systeme

Service-orientierte Systeme bestehen aus einem System von Services. Diese Services stellen über Schnittstellen bestimmte Funktionalitäten zur Verfügung, die wiederum von Services in diesem System genutzt werden können. Das ermöglicht die schnelle Entwicklung von komplexen Softwaresystemen mit umfangreichen Funktionen. Im Folgenden sollen verschiedene Schlagwörter im Bereich der Services und Service-orientierter Systeme und Services und Service-orientierte Systeme selbst, definiert werden.

3.1.1. Services

Die Implementierung eines Services wird als Π bezeichnet und enthält alle vom Service implementierten Prozeduren. Die Menge aller implementierten Prozeduren und Funktionen, die ein Service S selbst implementiert und nicht selbst implementiert aber aufruft, wird als $PROC_S$ bezeichnet.

Definition 3.1 (Service)**.** Ein Service sei ein Tripel $S = (\Pi, \mathbb{I}_S, \mathbb{R}_S)$ mit

- $\Pi \subseteq PROC_S$ die Implementierung der Prozeduren $PROC_S$ von Service S ist,
- $\mathbb{I}_S$ die endliche Menge von Angebotsschnittstellen, die von anderen Services aufgerufen werden können und
- $\mathbb{R}_S$ ist die endliche Menge der Nutzungsschnittstellen, die von Service S benötigt werden.

Definition 3.2 (Klientenservice/ Initialer Service)**.** Ein Klientenservice M ist ein Service, der in der Menge $PROC_M$ die *main*-Methode enthält.

Schnittstellen

Eine Schnittstelle beziehungsweise die Schnittstellenbeschreibung ist eine endliche Menge an Prozedur- oder Funktionsbeschreibungen, die von einem Service benötigt oder angeboten werden und aus der Menge $PROC$ sind. In dieser Arbeit werden Signaturen im C-Format/Notation angegeben.

Ein Service X kann eine Schnittstelle I_X zur Verfügung stellen, wobei I_X eine Angebotsschnittstelle ist. Stellt ein Service Prozeduren bereit, so müssen diese von dem Service X implementiert werden. Diese Implementierung kann wiederum andere Prozeduren oder Funktionen anderer Services aufrufen. Die Menge der Signaturen dieser anderen aufgerufenen Prozeduren und Funktionen werden Nutzungsschnittstellen R_X von X genannt.

Definition 3.3 (Angebotsschnittstelle)**.** Eine Angebotsschnittstelle (*engl. provided interface*) eines Service X ist eine Menge $I_X \subseteq sig$ ist, wobei sig die Menge der Signaturen der im Service X implementierten Prozeduren $p \in PROC$ ist.

Definition 3.4 (Nutzungsschnittstellen)**.** Eine Nutzungsschnittstelle vom Service X (*engl. required interface)* ist eine Menge R_X von Signaturen, wobei $R_X \cap sig = \emptyset$ für die Menge sig der Signaturen der im Service X implementierten Prozeduren $p \in PROC$ ist.

Das heisst, dass die Signaturen in R_X nicht in X implementiert sind und daher benötigt werden.

Definition 3.5 (Signatur, nach [10])**.** Eine Signatur sig ist definiert als ein Paar $sig = (N, O)$, wobei

$N \in Ident_N$	ist der Name mit $Ident_N \subseteq \Sigma^+$,
$O \subseteq Data \times Ident_O$	ist eine Liste von Parametern der Signatur, wobei sich die Namen i der Parameter unterscheiden müssen $\forall (d_k, i_k), (d_m, i_m) \in O, \forall k, m \in \mathbb{N}$ gilt, wenn $k \neq m \Rightarrow i_k \neq i_m$, wobei $Ident_O \subseteq \Sigma^*$ und $Data$ ist die endliche Menge von Datentypen.

Bemerkung 3.1 (Datentyp)**.** *In dieser Arbeit werden ausschließlich die Datentypen $BOOL$ und INT verwendet.*

Demnach sei der Service S in Abbildung 3.1 definiert durch das Tripel $S = (\Pi, \mathbb{I}_S, \mathbb{R}_S)$, wobei:

- Π ist die Implementierung der Prozedur a,
- $\mathbb{I}_S = \{I_S\}$ und
- $\mathbb{R}_S = \{R_S\}$.

Der Service S aus Abbildung 3.1 ist kein Klientenservice.

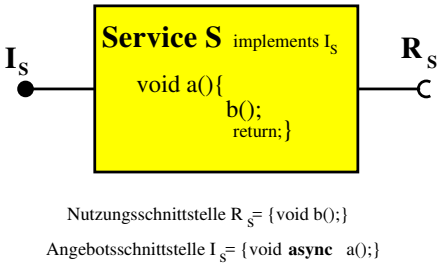


Abbildung 3.1.: Ein Service S mit der Angebotsschnittstelle I_S und der Nutzungsschnittstelle R_S.

3.1.2. Service-orientierte Systeme

Durch die Komposition beziehungsweise die Nutzung der Angebotsschnittstellen von Services durch Services entsteht über die Verknüpfung der Angebots- und Nutzungsschnittstellen der Services ein Service-orientiertes System.

Definition 3.6 (Service-orientiertes System)**.** Sei SoS ein Service-orientiertes System mit $SoS = (S_{sos}, \mathbb{I}_{S_{sos}}, \mathbb{R}_{S_{sos}})$, wobei

- S_{sos} eine endliche Menge an Services ist,
- $\mathbb{I}_{S_{sos}}$ die endliche Menge aller Nutzungsschnittstellen im SoS ist und
- $\mathbb{R}_{S_{sos}}$ die endliche Menge aller Angebotsschnittstellen im SoS ist.

Komposition

Definition 3.7 (Komposition)**.** Seien $S_1, \cdots, S_n$ die Services mit Implementierungen Π, Angebotsschnittstellen $\mathbb{I}_{\sqsupset}$ und Nutzungsschnittstellen $\mathbb{R}_{\sqsupset}$, dann ist $SoS \widehat{=} (S, \mathcal{B}_{sos}, \mathbb{I}_{S_{sos}}, \mathbb{R}_{S_{sos}})$ wobei:

- $S_{sos} = \{S_1 \cup \cdots \cup S_n\}$ für ein $n \in \mathbb{N}$,
- $\mathcal{B}_{sos}$ ist die Menge der Bindungen,
- $\mathbb{I}_{S_{sos}} = \{\mathbb{I}_{S_1} \cup \cdots \cup \mathbb{I}_{S_n}\}$
- $\mathbb{R}_{S_{sos}} = \{\mathbb{R}_{S_1} \cup \cdots \cup \mathbb{R}_{S_n}\}$

Definition 3.8 (Bindung, nach [10])**.** Sei $\mathcal{B}$ die Menge der Bindungen zwischen Services eines Service-orientierten Systems SoS, wobei $\mathcal{B} \widehat{=} \{(S_1, S_2) : S_1 \in S_{sos} \wedge S_j \in S_{sos} \wedge I \in (\mathbb{I}_{S_j} \cap \mathbb{R}_{S_i})\}$ ist.

3.1.3. Ausführungsmodell

Prozeduren oder Funktionen können synchron oder asynchron aufgerufen werden.

Definition 3.9 (synchrone Aufrufe [10])**.** Wird eine Prozeduren oder Funktionen einer Angebotsschnittstelle synchron aufgerufen, so muss der aufrufende Service warten, bis die aufgerufenen Prozedur oder Funktion des aufgerufenen Service ausgeführt wurde und zurückkehrt. Bis zu diesem Zeitpunkt kann der aufrufende Service den nachfolgenden Programmcode nicht ausführen (ausgeschlossen sind eventuell bereits asynchron laufende Aufrufe). Aus diesem Grund werden synchrone Aufrufe auch als blockierende Aufrufe bezeichnet.

Definition 3.10 (asynchrone Aufrufe [10])**.** Wird eine Prozedur oder Funktion einer Angebotsschnittstelle asynchron aufgerufen, so können der aufrufende Service und der aufgerufene Service mit der Ausführung fortfahren, bis die *return*-Anweisung der aufgerufenen und aufrufenden Prozedur oder Funktion erreicht wird. Dann muss, bevor der Aufrufer mit der Ausführung fortfährt, mit der asynchron aufgerufenen Prozedur oder Funktion synchronisieren. Asynchrone Prozeduren oder Funktionen bekommen den Zusatz **async** in der jeweiligen Schnittstellenbeschreibung.

Bemerkung 3.2. *Bei der Komposition ist zu beachten, dass es asynchrone und synchrone Prozeduren und Funktionen in den Angebotsschnittstellen und Nutzungsschnittstellen gibt. Es ist nicht möglich, dass asynchrone an synchrone Prozeduren oder Funktionen binden und umgekehrt.*

Definition 3.11 (Synchronisation)**.** In dem Ausführungsmodell dieser Arbeit existieren zwei Möglichkeiten asynchrone Prozeduren oder Funktionen zu synchronisieren. Die erste Möglichkeit besteht darin, dass der Aufrufende eine *sync*-Anweisung erreicht (vgl. mit Syntax A.6 im Anhang A). Hier wird die Ausführung des Aufrufers blockiert und auf die Rückkehr der aufgerufenen Prozedur oder Funktion gewartet. Die zweite Möglichkeit ist die Synchronisation des Aufrufer mit dem Aufgerufenen beim Erreichen der *return*-Anweisung in beiden Prozeduren oder Funktionen (siehe Def. 3.10).

3.2. Formale Modelle

Zur Modellierung von Softwaresystemen können verschiedene Modelle benutzt werden. Im Folgenden sollen die für die Arbeit wichtigen Modelle kurz vorgestellt werden.

3.2.1. Petri-Netze

Die Theorie der Petri-Netze hat ihren Ursprung in der Dissertation von Carl A. Petri [44] und wurden im Laufe der Zeit durch Arbeiten wie [48] erweitert.

Definition 3.12 (Petri-Netz)**.** Ein Petri-Netz P ist ein Tupel $P = (S, T, A, \lambda)$ mit

- $S = \{s_1, \ldots, s_{|S|}\}$ einer endlichen Menge an Stellen,
- $T = \{t_1, \ldots, t_{|T|}\}$ einer endlichen Menge an Transitionen, $S \uplus T$ $(S \cap T = \emptyset)$,
- $A \subseteq \mathrm{S} \times \mathrm{T} \cup \mathrm{T} \times \mathrm{S}$ (arcs) und
- $\lambda : A \to \mathbb{N}$ eine Markierungsfunktion.

Ein Petri-Netz P kann graphisch als bipartiter Graph $G = (E, V)$ dargestellt werden, wobei $E = S \cup T$ die Ecken sind und die Kanten V die Relationen in A entsprechen. Stellen werden als Kreise, Transitionen als Rechtecke und die Relationen als Kanten mit Pfeilen dargestellt (gerichtet).

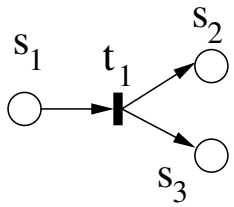


Abbildung 3.2.: Ein Petri-Netz mit 3 Stellen und einer Transition.

In Abbildung 3.2 ist ein Petri-Netz zu finden. Das Petri-Netz besteht aus 3 Stellen s_1, s_2 und s_3 sowie aus einer Transition t_1.

Definition 3.13 (Vor- und Nachbereich)**.** Sei $x \in S \cup T$ so definieren wir $\bullet x := \{y \in S \cup T : (y, x) \in S \times T\}$ als Vorbereich von x und $x\bullet := \{y \in S \cup T : (x, y) \in S \times T\}$ als Nachbereich von x.

Definition 3.14 (Vorstellen und Nachstellen)**.** Die Menge der *Vorstellen* einer Transition t ist definiert als $Pre(t) = \{p : (p, t) \in E\}$. Analog, ist die Menge der *Nachstellen* von t definiert als $Post(t) = \{p : (t, p) \in E\}$.

Definition 3.15 (Zustandsraum)**.** Sei $P = (S, T, A)$ ein Petri-Netz. Der Zustandsraum eines Petri-Netzes ist $\mathbb{N}^{|S|}$. Die Funktion $\mu : S \to \mathbb{N}$ ist ein Zustand oder eine Markierung von P. Markierungen werden als Spaltenvektoren angegeben.

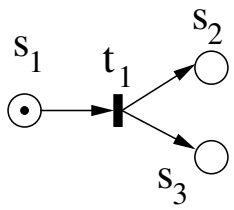


Abbildung 3.3.: Das Petri-Netz aus Abbildung 3.2 mit einer Marke in der Stelle s_1.

Wenn $\mu(s_i) = k_i$ hat die Stelle s_i die Anzahl k_i Marken in der Markierung μ. Die Marken werden graphisch als schwarze Punkte in den Stellen angegeben. In Abb. 3.3 ist das Petri-Netz aus Abb. 3.2 mit einer Marke in der Stelle s_1 zu sehen. Die zu dem Zustand dazugehörige Markierung lautet $\mu = (1, 0, 0)^T$.

Definition 3.16 (Anfangsmarkierung)**.** Eine initiale Markierung oder Anfangsmarkierung ist definiert als Spaltenvektor μ_0 wobei gilt:

- $\mu(s_0) = 1$ mit $s_0 \in S$,
- $\mu(s_f) = 0$ mit $s_f \in S$ und
- $\forall s_i \in S : \mu(s_i) = 0,\ i \in \mathbb{N}$.

Definition 3.17 (Finalmarkierung)**.** Eine Finalmarkierung ist definiert als Spaltenvektor μ_f wobei gilt:

- $\mu(s_f) = 1$ and
- $\forall s_i \in S : \mu(s_i) = 0,\ i \in \mathbb{N}$.

Definition 3.18 (Initiales Petri-Netz)**.** Ein initiales Petri-Netz P ist gegeben als $P = (S, T, A, \lambda, \mu_0)$ von einem

- Petri-Netz $P = (S, T, A, \lambda)$ und
- einer Anfangsmarkierung $\mu_0 \in \mathbb{N}$.

Wenn der Zustand des Petri-Netzes in Abbildung 3.3 das initiale Petri-Netz repräsentiert, dann lautet die Anfangsmarkierung für dieses Petri-Netz $\mu_0 = (1, 0, 0)^T$.

Definition 3.19 (Geladene Transitionen)**.** Sei P ein initiales Petri-Netz mit $P = (S, T, A, \lambda, \mu_0)$ und einer Transition $t \in T$. Die Transition t ist im Zustand μ geladen, wenn $\mu(s) \geq \lambda((p,t))$ für alle $p \in Pre(t)$ gilt.

Die Transition t_1 im Petri-Netz von Abbildung 3.3 ist nach dieser Definition geladen. Wenn eine Transition geladen ist, dann kann sie feuern.

Definition 3.20 (Feuern)**.** Sei P ein initiales Petri-Netz mit $P = (S, T, A, \lambda, \mu_0)$ und einer Transition $t \in T$. Wenn die Transition t feuert, dann kann der Folgezustand μ' von μ folgendermaßen berechnet werden:

$$\mu'(p) \widehat{=} \begin{cases} \mu(p) + \lambda(t,p), & \text{falls } p \in Post(t) \setminus Pre(t) \\ \mu(p) - \lambda(p,t), & \text{falls } p \in Pre(t) \setminus Post(t) \\ \mu(p) - \lambda(p,t) + \lambda(t,p), & \text{falls } p \in Pre(t) \cap Post(t) \\ \mu(p), & \text{sonst} \end{cases}$$

Deadlockanalyse

Bei der Deadlockanalyse werden Petri-Netze untersucht, ob sie immer von dem Zustand der Anfangsmarkierung in den Zustand der Finalmarkierung gelangen. Ist dies nicht der Fall, so existiert ein Deadlock.

Definition 3.21 (Deadlock)**.** Ein Deadlock d ist ein Zustand eines initialen Petri-Netzes $P = (S, T, A, \lambda, \mu_0)$ mit einem Finalzustand μ_f, wenn $d \neq \mu_f$ ist und keine weitere Transition $t \in T$ geladen ist.

3.2.2. Die PRS-Hierarchie

Mayr präsentierte eine vereinte Darstellung von Petri-Netzen und verschiedenen einfachen Prozessalgebren, in dem er sie als Subklassen der generellen *rewriting* Formalismus *Process Rewrite Systems* präsentiert [39]. Dieser Formalismus basiert auf Termersetzungsregeln auf prozess-algebraischen Ausdrücken. Die Menge $PEX(Q)$ von prozess algebraischen Ausdrücken über eine endliche Menge Q (*atomic processes* von atomaren Prozessen ist die kleinste Menge, die folgende Bedingungen erfüllt:

(i) $\varepsilon \in PEX(Q)$

(ii) $Q \subseteq PEX(Q)$,

(iii) Wenn $e, e' \in PEX(Q)$, dann $e.e' \in PEX(Q)$ und $e \parallel e' \in PEX(Q)$ (entsprechend sequentieller und paralleler Komposition).

$$\frac{e \to e'}{e \Rightarrow e'} \text{ (R)} \qquad \frac{e \Rightarrow e'}{e.s \Rightarrow e'.s} \text{ (S)} \qquad \frac{e \Rightarrow e' \quad e' \Rightarrow e''}{e \Rightarrow e''} \text{ (T)}$$

$$\frac{e \Rightarrow e'}{e \parallel s \Rightarrow e' \parallel s} \text{ (P1)} \qquad \frac{e \Rightarrow e'}{s \parallel e \Rightarrow s \parallel e'} \text{ (P2)} \qquad \frac{}{u \Rightarrow u} \text{ (L)}$$

$$e, e', e'', s \in PEX(Q)$$

Abbildung 3.4.: Inferenzregeln für die Definition der Ableitungsrelation in einem PRS

Die parallele Komposition ist assoziativ und kommutativ. Die sequentielle Komposition ist assoziativ aber nicht kommutativ. Der leere atomare Prozess ε ist das neutrale Element bzgl. der sequentiellen und parallelen Komposition.

Definition 3.22 (Prozessersetzungssystem)**.** Ein Prozessersetzungssystem (engl. process rewrite system; Abk.: PRS) ist ein Tupel $\Pi \triangleq (Q, q_0, \to, F)$ wobei

(i) Q eine endliche Menge *atomarer Prozesse* ist,

(ii) $q_0 \in Q$ (ein *initialer Zustand*, atomarer Prozess),

(iii) $\to \subseteq PEX(Q) \times PEX(Q)$ eine Menge von *Prozess-Ersetzungsregeln* ist und

(iv) $F \subseteq Q$ eine Menge *finaler Prozesse* ist.

Das PRS Π definiert eine Ableitungsrelation $\to \subseteq PEX(Q) \times PEX(Q)$ über die Inferenzregeln aus Abb. 3.4.

Ein PRS, dessen Ersetzungsregeln keine sequentiellen Operatoren "." enthalten, sind die (P,P)-PRS, die äquivalent zu Petri-Netzen sind [39]. Eine Übersicht über die PRS-Hierarchie ist in Abb. A.2 im Anhang A zu finden. Aus diesem Grund, sind die folgenden Definitionen auf generelle (G,G)-PRS und auch auf Petri-Netze anwendbar.

Definition 3.23 (Deadlock in PRS)**.** Sei $\Pi = (Q, q_0, \to, F)$ ein PRS. Ein prozessalgebraischer Ausdruck $e \in PEX(Q)$ ist genau dann *erreichbar*, wenn $q_0 \Rightarrow e$. Ein erreichbarer prozess-algebraischer Zustand $e \in PEX(Q)$ ist ein *Deadlock* genau dann wenn kein prozess-algebraischer Ausdruck $e' \in PEX(Q) \setminus F$ existiert, $e' \neq e$ so dass $e \Rightarrow e'$.

3.2.3. Kaktuskeller

Kaktuskeller wurden als Bäume von Kellern durch [19] eingeführt und auf die Arbeiten von [28]. Das Ausführungsmodell dieser Arbeit beinhaltet sequentielles Verhalten sowie paralleles Verhalten. Beide Verhaltensweisen können von Kaktuskellern dargestellt werden.

Definition 3.24 (Kaktuskeller, [10])**.** Ein Kaktuskeller [28], [19] ist eine Kellervariante bei der an das oberste Kellerelement ein weiterer Keller verlinkt werden kann. Diese verlinkten Keller heißen Äste. Alle obersten Kellerelemente können bearbeitet werden, darunter liegende Kellerelement können nicht bearbeitet werden. Wenn ein Ast leer ist, kann dieser entfernt werden.

Ein Programmzustand kann mit einem Kaktuskeller dargestellt werden. Dabei wird jeder Prozess von einem Keller repräsentiert. Wenn eine asynchrone Prozedur oder Funktion aufgerufen wird, wird ein Kellerelement auf den Keller gelegt. Das neue Kellerelement repräsentiert den aufrufenden Prozess. Wenn eine asynchrone Prozedur aufgerufen wird, wird ein neuer Keller erzeugt, der mit dem Kellerelement verlinkt wird. Der daraus resultierende Kaktuskeller ähnelt den Kakteen *Carnegiea gigantea*, siehe Abb. B.1 und daher auch die entsprechende Namensgebung. Wird also ein Programmpunkt ausgeführt, wird in der Darstellung als Kaktuskeller, ein Kaktuskeller in einen anderen Kaktuskeller transformiert.

Eine Zusammenfassung einiger Kontrollstrukturen und deren Modellierung mit Kaktuskellern ist in Tab. 3.1 zu sehen.

(G,G)-PRS und Kaktuskeller

Nach [29] können Zustände und Zustandsübergänge von (G,G)-PRS mithilfe von Kaktuskellern grafisch dargestellt werden. Wird ein System mithilfe eines (G,G)-PRS nach Tabelle 3.2 modelliert, so kann ein Lauf in diesem System mithilfe der dazugehörigen Kaktuskeller dargestellt werden (siehe Tab. 3.1).

3.3. Abstraktion des Programmverhaltens

Um das Verhalten eines Programms modellieren zu können, wird von dem vorliegenden Programm abstrahiert. Im Prinzip gehört jede Anweisung zu einem Programmpunkt. Wichtige Kontrollstrukturen sind atomare Anweisungen, wie zum Beispiel Zuweisungen, bedingte Anweisungen, synchrone Prozedur- oder Funktionsaufrufe, asynchrone Prozedur- oder Funktionsaufrufe oder Synchronisationen. Im nächsten Unterabschnitt wird die Modellierung monolithischer Programme mithilfe eines Abstraktionsverfahren nach [11] vorgestellt.

```
...
void main() {
    a();
    //something(no call/sync)
    return;
}
void a() { //Prozedur a ist synchron
    if e1
        b();
    else
        c();
    return;
}
void b() { //Prozedur b ist asynchron
    //something(no call/sync)
    return;
}
void c(){ //Prozedur c ist asynchron
    //something(no call/sync)
    return;
}
...
```

Abbildung 3.5.: Ein kleines Beispielprogramm.

3.3.1. Modellierung monolithischer Programme

Zur Modellierung eines Programms wird das Abstraktionsverfahren von unterschiedlichen Kontrollstrukturen eines Programms zu (P,P)- und (G,G)-PRS-Regeln in Tab. 3.2 gezeigt (nach [11]). Durch die Anwendung der Regeln in dieser Tabelle entsteht eine (G,G)-PRS $\Pi_{(G,G)-PRS} = (Q, \rightarrow, q_0, F)$. Die Regelmenge $\rightarrow$ ergibt sich aus der schrittweisen Anwendung der Regeln aus der Tabelle 3.2 auf jeden Programmpunkt des zu modellierenden Programms. Der Zustand q_0 ist der erste Programmpunkt im Rumpf der *main*-Methode und F ist die einelementige Menge, die den Programmpunkt der *return*-Anweisung von *main* enthält.

Beispiel 3.1. Zur Verdeutlichung des Abstraktionsverfahren ist in Abbildung 3.5 der Quellcode eines kleinen Beispielprogramms zu sehen. Die Prozedur a ist synchron und die Prozeduren b und c sind asynchron. Die Kennzeichnung ist hier nur als Kommentar eingefügt, da die Arbeit sich nicht mit der Abstraktion monolithischer Programme beschäftigt. Prozeduren und Funktionen, die über Systemgrenzen hinaus aufgerufen werden, werden in der Schnittstellenbeschreibung klassifiziert.

Aus diesem Programm ergeben sich nach Anwendung der Abstraktionsregeln in Tab. 3.2 die Regeln der (G,G)-PRS-Abstraktion in Abb. 3.6.

3.3.2. Modellierung von Service-orientierten Systemen

Um die Abstraktionen für ein Service-orientiertes System zu erstellen, wird für jede Prozedur p in der Angebotsschnittstelle I_S von jedem Service S eine Abstraktion durch

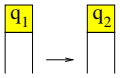
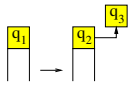
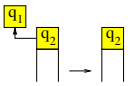
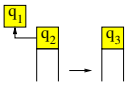
Kontrollstruktur	Kaktuskeller	Kontrollstruktur	Kaktuskeller
Zuweisung (PSR)		asynchroner Prozeduraufruf	
synchroner Prozeduraufruf		return des asynchronen Prozeduraufruf	
return synchroner Prozeduraufrufs		Synchronisierung	

Tabelle 3.1.: Kontrollstruktur und dazugehörige Darstellung des Kaktuskellers.

Quellcode aus Abbidlung. 3.5	Regeln des (G,G)-PRS

main{
q_0 : $a(\dots)$;
q_1 : //*something*(*no call/sync*)
q_f : **return**
}
	$q_0 \rightarrow q_2.q_1$
	$q_1 \rightarrow q_f$

$a(\dots)$**{**
q_2 : **if** e_1
q_3 : $b(\dots)$;
else
q_4 : $c(\dots)$;
q_5 : **return;**
}
	$q_2 \rightarrow q_3$
	$q_2 \rightarrow q_4$
	$q_3 \rightarrow q_5 \parallel q_6$
	$q_4 \rightarrow q_5 \parallel q_8$
	$q_5.q_f \rightarrow q_f$

$b(\dots)$**{** q_6 : *something*(*no call/sync*) q_7 : **return;** }	$q_6 \rightarrow q_7$ $q_7 \parallel q_5 \rightarrow q_5$
$c(\dots)$**{** q_8 : *something*(*no call/sync*) q_9 : **return;** }	$q_8 \rightarrow q_9$ $q_9 \parallel q_5 \rightarrow q_5$

Abbildung 3.6.: Abstraktion des Quellcodes von Abbildung 3.5 zu (G,G)-PRS-Regeln.

Kontrollstrukturen	Abstraktion	Beschreibung
q_i : *Zuweisung*; q_j : ···	**(G,G)** $q_i \rightarrow q_j$ **(P,P)** $q_i \rightarrow q_j$	Zuweisungen haben keinen Einfluss auf das Deadlock-Verhalten. q_j ist der Programmpunkt mit der Anweisung, die nach der Zuweisung im Programmpunkt q_i ausgeführt wird.
q_i : **while** e{ q_j : ··· q_k : } q_l : ···	**(G,G)** $q_i \rightarrow q_j$ $q_k \rightarrow q_l$ $q_i \rightarrow q_l$ **(P,P)** $q_i \rightarrow q_j$ $q_k \rightarrow q_l$ $q_i \rightarrow q_l$	Die Auswertung der Bedingung e hat keinen Einfluss auf das Deadlock-Verhalten. q_j ist der Programmpunkt der Anweisung, die ausgeführt wird, wenn e zu $wahr$ ausgewertet wird. q_l ist der Programmpunkt der Anweisung, die ausgeführt wird, wenn e zu $falsch$ ausgewertet wird.

q_i : **if** e{
q_j : ···
q_k : }
else{
q_l : ···
q_m : }
q_n : ···
	(G,G)
	$q_i \rightarrow q_j$
	$q_i \rightarrow q_l$
	$q_k \rightarrow q_n$
	$q_m \rightarrow q_n$
	(P,P)
	$q_i \rightarrow q_j$
	$q_i \rightarrow q_l$
	$q_k \rightarrow q_n$
	$q_m \rightarrow q_n$
		Keinen Einfluss auf das Deadlockverhalten haben bedingte Anweisungen. q_n ist der Programmpunkt der Anweisung, die nach Verlassen des if- oder $else$-Zweigs ausgeführt wird (q_k oder q_m).

Synchronisation q_i : **sync** b(); q_{i+1} : ··· b{ ··· q_j : **return;**}	**(G,G)** $q_i \parallel q_j \rightarrow q_{i+1}$ **(P,P)** $q_i \parallel q_j \rightarrow q_{i+1}$	Die Anweisung nach q_i (Programmpunkt q_{i+1}) kann nur ausgeführt werden, wenn die vorherige asynchron aufgerufene Prozedur b die $return$-Anweisung erreicht (Programmpunkt q_j).
Synchrone Prozedur a q_i : a(); q_{i+1} : ··· a{ q_j : ··· q_k : **return;**}	**(G,G)** $q_i \rightarrow q_j \cdot q_{i+1}$ $q_k \cdot q_{i+1} \rightarrow q_{i+1}$ **(P,P)** $q_i \rightarrow q_j$ $q_k \rightarrow q_{i+1}$	Aufruf der synchronen Prozedur a: Der Programmpunkt q_{i+1} des Programmpunktes, der nach dem Aufruf von a aufgeführt werden soll, wird auf den Keller gelegt. Die Ausführung wird am ersten Programmpunkt q_j von a ausgeführt.
Asynchrone Procedur b a{ ··· q_i b(); q_{i+1} ··· q_j **return;** } ··· b{ q_k : ··· q_l : **return;**}	**(G,G)** $q_i \rightarrow q_{i+1} \parallel q_k$ $q_j \parallel q_l \rightarrow q_j$ **(P,P)** $q_i \rightarrow q_{i+1} \parallel q_k$ $q_j \parallel q_l \rightarrow q_j$	Aufruf einer asynchronen Prozedur b: Die Ausführung kann nebenläufig am Programmpunkt q_{i+1} nach dem Aufruf und bei der ersten Anweisung am ersten Programmpunkt q_k der asynchronen Prozedur b erfolgen.

Tabelle 3.2.: Kontrollfluss-basierte Abstraktion zu (G,G)-PRS und (P,P)-PRS [29].

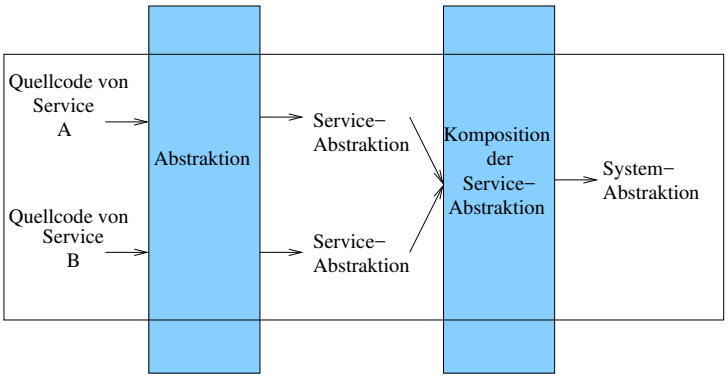


Abbildung 3.7.: Abstraktionsprozess.

Anwendung der (G,G)-PRS-Regeln aus Tabelle 3.2 erzeugt.

Service-Abstraktionen

Der Hauptunterschied zu monolithischen Programmen besteht darin, dass die Ein- und Ausstiegspunkt (i_p und r_p) einer jeden in der Schnittstelle vorhandene Prozedur p aufgerufen werden muss. Damit ergibt sich folgende Definition für die Service-Abstraktionen:

Definition 3.25 (Prozedur-Abstraktion)**.** Sei p eine Prozedur aus der Angebotsschnittstelle I_S vom Service S. Dann ist die (G,G)-PRS Abstraktion definiert durch $[I_{S_p}] \widehat{=} (Q_{I_{S_p}}, q_{0_{I_{S_p}}}, \rightarrow_{I_{S_p}}, F_{I_{S_p}})$, wobei

- $Q_{I_{S_p}} \subseteq PEX(Q_{I_{S_p}})$
- $q_{0_{I_{S_p}}} = i_p$, wobei i_p der erste Programmpunkt in Rumpf von p ist,
- $\rightarrow_{I_{S_p}}$, die Regeln, die durch Anwendung der Tabelle 3.2 auf den Quellcode entstehen und
- $F_{I_{S_p}} = \{r_p\}$, wobei r_p der Programmpunkt der *return*-Anweisung von p ist.

Definition 3.26 (Schnittstellen-Abstraktion)**.** Sei I_S eine Angebotsschnittstelle von Service S und $p_0, \ldots, p_n$ mit $n \in \mathbb{N}$ die Menge der angebotenen Prozeduren, dann sei die (G,G)-PRS-Abstraktion dieser Schnittstelle $[I_S] = (Q_{I_S}, Q_{0_{I_S}}, \rightarrow_{I_S}, F_{I_S})$, wobei:

- $Q_{I_S} = \bigcup_{i=0}^{n} Q_{I_{S_{p_i}}}$,

- $Q_{0_{I_S}} = \bigcup_{i=0}^{n} \{q_{0_{I_{S_{p_i}}}}\}$,
- $\rightarrow_{I_S} = \bigcup_{i=0}^{n} \rightarrow_{I_{S_{p_i}}}$ und
- $F_{I_S} = \bigcup_{i=0}^{n} F_{I_{S_{p_i}}}$.

Aus der Definition 3.26 der Schnittstellen-Abstraktion ergibt sich die Definition der Service-Abstraktionen:

Definition 3.27 (Service-Abstraktion)**.** Seien $I_{S_0}, \ldots, I_{S_n}$ mit $n \in \mathbb{N}$ die Angebotsschnittstellen eines Service S, dann sei die (G,G)-PRS-Abstraktion des Service S $[S] = (Q_S, Q_{0_S}, \rightarrow_S, F_S, (\mathcal{P})_S)$ mit:

- $Q_S = \bigcup_{i=0}^{n} Q_{I_{S_i}}$,
- $Q_{0_S} = \bigcup_{i=0}^{n} Q_{0_{I_{S_i}}}$,
- $\rightarrow_S = \bigcup_{i=0}^{n} \rightarrow_{I_{S_i}}$,
- $F_S = \bigcup_{i=0}^{n} F_{I_{S_i}}$,
- $(\mathcal{P})_S : E \rightarrow 0_S$ ist eine Abbildungsfunktion auf den ersten Programmpunkt aller Prozeduren p in allen Angebotsschnittstellen von S.

3.3.3. Kompositionsverfahren

Bei der Komposition der Service-Abstraktionen müssen nun alle Service-Abstraktionen vereinigt werden. Die Programmpunkte der Servicegrenzen-übergreifenden Prozedur- und Funktionsaufrufe eines Services werden mit den Abbildungsfunktionen $(\mathcal{P})_S$ aufgelöst. Damit ergibt sich folgende Definition für die System-Abstraktion:

Definition 3.28 (System-Abstraktion)**.** Seien $S_0, \ldots, S_n$ mit $n \in \mathbb{N}$ die Services und $K_0, \ldots, K_k$ mit $k \in \mathbb{N}$ die Klientenservices in einem Service-orientierten System SoS, dann sei die (G,G)-PRS-Abstraktion dieses Systems SoS
$[SoS] = (Q_{SoS}, Q_{0_{SoS}}, \rightarrow_{SoS}, F_{SoS}, (\mathcal{P})_{SoS})$ mit:

- $Q_{SoS} = \bigcup_{i=0}^{n} Q_{S_i}$,
- $Q_{0_{SoS}} = \bigcup_{i=0}^{n} Q_{0_{K_i}}$ für alle Klientenservices,
- $\rightarrow_{SoS} = \bigcup_{i=0}^{n} \rightarrow_{S_i}$ und
- $F_{SoS} = \bigcup_{i=0}^{n} F_{K_i}$.

3.3.4. Verifikation von Systemmodellen

Definition 3.29 (Verifikation von Systemmodellen)**.** Es liege ein Systemmodell (z.B. Kellerautomaten oder Petri-Netze) und eine Spezifikation einer Eigenschaft vor. Bei der Verifikation wird geprüft, ob diese Eigenschaft (z.B. Deadlockfreiheit) vom Systemmodell erfüllt wird oder nicht. Jeder Teil des Systemmodells, dass nicht die Eigenschaft erfüllt, wird Gegenbeispiel genannt.

Ein Beispiel für eine Verifikation ist die Deadlockanalyse auf Petri-Netzen.

Definition 3.30 (Falsch Positive bei System-Abstraktionen)**.** Es liegen falsch positive Ergebnisse vor, wenn das Systemmodells die gewünschte Eigenschaft nicht erfüllt und obwohl das System, von dem abstrahiert wurde, die Eigenschaft vorliegt.

Die Deadlockanalyse bei einem Systemmodell basierend auf Petri-Netzen wird durchgeführt. Es liegt ein falsch positives Ergebnis vor, wenn das vorhandene Petri-Netz als deadlockfrei klassifiziert wird, obwohl ein Deadlock im System vorhanden ist.

Definition 3.31 (Falsch Negative bei System-Abstraktionen)**.** Es liegen Falsch Negative Ergebnisse vor, wenn das Systemmodells die gewünschte Eigenschaft erfüllt und diese Eigenschaften aber nicht im System, von dem abstrahiert wurde, vorliegt. Diese Ergebnisse werden auch unechte Gegenbeispiele (engl. spurious counterexamples) genannt.

3.4. Zusammenfassung

In diesem Kapitel wurden die Grundlagen, auf die diese Arbeit aufbaut, erläutert. Neben den vorgestellten Modellen Petri-Netze, PRS und Kaktuskeller gibt es natürlich noch andere Modelle zur Systemmodellierung, auf die aber in dieser Arbeit nur im Kapitel 2,

die Verwandte Arbeiten, eingegangen wird und deswegen hier nicht vorgestellt werden. Im folgenden Kapitel wird die Beispielsprache $\mathcal{XYZ}$ vorgestellt und die operationale Semantik, die das Verhalten der Sprache definiert, vorgestellt.

4. Operationale Semantik

Im folgenden Kapitel wird auszugsweise die Beispielsprache $\mathcal{XYZ}$ und eine operationale Semantik zur Beschreibung des Verhaltens dieser Beispielsprache vorgestellt. Die operationale Semantik wird erweitert, um auch das Verhalten von Service-orientierten Systemen aufzeigen zu können. Es ist das Ziel, Programme oder Service-orientierte Systeme dieser Beispielsprache auf bestimmte Eigenschaften zu untersuchen. Dabei liegt der Fokus auf der Deadlockanalyse von monolithischen oder Service-orientierten Softwaresystemen.

4.1. Die Beispielsprache $\mathcal{XYZ}$

In diesem Abschnitt wird informell die Beispielsprache $\mathcal{XYZ}$ vorgestellt. Sie beschränkt sich auf grundlegende Konzepte. Für diese Grundkonzepte wird eine operationale Semantik definiert. Obwohl eine Semantik auf der abstrakten Syntax beruht, wird im Folgenden die konkrete Syntax definiert, um überhaupt Programme beziehungsweise Service-orientierte Systeme formulieren zu können.

4.1.1. Syntax

Die Syntax liegt als kontextfreie Grammatik in *EBNF* vor. Die ausführliche Beschreibung der Beispielsprache $\mathcal{XYZ}$ ist im Anhang A nachzulesen.

Vordefinierte Bezeichner

Die Beispielsprache besitzt folgende vordefinierte Bezeichner:

Bezeichner	Bedeutung
int	Integer Typ.
bool	Logischer Typ.
false	Falschheit.
true	Wahrheit.

Mögliche Typen in der Beispielsprache sind int und bool.

Anweisungen

Die Syntax von Anweisungen sei hier durch einen kleinen Ausschnitt als kontextfreie Grammatik in *EBNF* gegeben:

```
Stat  ::= Expr | Iteration | Assign | Sync | ProcC | Skip
...
Cond  ::= "if" Expr "{" Stats "}" "else" "{" Stats "}"
    | "if" Expr "{" Stats "}"
ProcC   ::= "call" Name "(" ")" ";"
    |"call" Name "(" argument ")" ";"
Sync  ::= "sync" Name ";"
...
```

Listing 4.1: Anweisungen in der Beispielsprache $\mathcal{XYZ}$ (Ausschnitt)

Bemerkung 4.1. *Die vorgestellte Sprache steht prototypisch für andere Sprachen und ist auf ein Minimum an Konzepten reduziert. Mit Hilfe dieser Grundkonzepte ist die Modellierung zustandsloser Services möglich. Konzepte wie Typfallunterscheidung, Ausnahmebehandlung oder OO-Polymorphie bauen auf diese Grundkonzepte auf und die Beispielsprache kann um diese Konzepte erweitert werden. Auch die Erweiterung um einen Speicher und Heap ist möglich [49], für diese Arbeit jedoch nicht von Interesse, da hier komplett von den Daten abstrahiert wird und die Reinkultur Service-orientierter Systeme betrachtet wird.*

4.1.2. Auswertung von Ausdrücken

Die Beispielsprache $\mathcal{XYZ}$ enthält Zuweisungen an Variablen, wie zum Beispiel x := y; und Zugriffe auf Variablen vom Typ int und bool. Ein lokaler Speicher für diese Variablen wird repräsentiert durch σ, der sich während der schrittweisen Ausführung des Programms von Programmpunkt zu Programmpunkt ändern kann. Das Programm gelangt durch die Ausführung der einzelnen Programmpunkte vom Startzustand in den Finalzustand. Dabei sei der Speicher wie folgt definiert:

Definition 4.1 (Speicher)**.** Sei $\sigma : VAR \to (INT \cup BOOL)$ die Speicherfunktion, die jeder Variable VAR einen Wert über INT (arithmetische Ausdrücke) oder $BOOL$ (boolesche Ausdrücke) zuordnet. σ_0 steht für den leeren Speicher, wobei $\sigma_0(x) = \perp$ (nicht definiert) für alle $n \in \mathbb{N}$, vgl. Tabelle B.1.

Der Inhalt eines Speichers σ wird durch [] notiert.

Beispiel 4.1. σ enthält die Programmvariablen $x \in VAR$ und $y \in VAR$, die beide vom Typ INT sind und beiden wurde der Wert 2 zugewiesen. Der Inhalt des Speichers von σ wird mit $[x := 2, y := 2]$ angegeben.

Definition 4.2 (Programmvariablen)**.** Die Menge der Programmvariablen eines Programms sei VAR.

Ausdrücke liefern einen Wert zurück. Der Wert kann dabei eine Zahl oder ein Wahrheitswert sein. Diese Werte verändern den Zustand nicht (zustandslose Services).

Semantik von Ausdrücken

Bei der Auswertung von Ausdrücken wird zwischen arithmetischen und booleschen Ausdrücken unterschieden.

Definition 4.3 (Semantik von arithmetischen Ausdrücken)**.** Die Semantik arithmetischer Ausdrücke a bezüglich eines Speichers σ ist durch $eval_A^\sigma(a)$ rekursiv definiert:

$$\begin{aligned} eval_A^\sigma(n) &= \mathcal{Z}(n). \\ eval_A^\sigma(x) &= \sigma(x) \\ eval_A^\sigma(a_1 + a_2) &= eval_A^\sigma(a_1) + eval_A^\sigma(b_2). \\ eval_A^\sigma(a_1 - a_2) &= eval_A^\sigma(a_1) - eval_A^\sigma(b_2). \end{aligned}$$

wobei n ein nummerisches Literal ist, $\mathcal{Z}(n)$ die ganze Zahl ($n \in \mathcal{Z}$) liefert und x eine arithmetische Programmvariable ist und $\sigma : VAR \to INT$ den Wert einer Programmvariable liefert.

Definition 4.4 (Semantik von booleschen Ausdrücken)**.** Die Semantik boolescher Ausdrücke b bezüglich eines Speichers σ ist durch $eval_B^\sigma(b)$ rekursiv definiert:

$$\begin{aligned} eval_B^\sigma(true) &= true. \\ eval_B^\sigma(not\, b) &= \neg eval_B^\sigma(b). \\ eval_B^\sigma(b_1 <= b_2) &= eval_B^\sigma(b_1) \leq eval_B^\sigma(b_2). \\ eval_B^\sigma(b_1 \,\&\&\, b_2) &= eval_B^\sigma(b_1) \wedge eval_B^\sigma(b_2). \end{aligned}$$

wobei b, b_1 und b_2 boolesche Ausdrücke sind.

Bemerkung 4.2. *Wenn in der Arbeit der Kontext klar hervorgeht, wird auf den Index zur Kennzeichnung des Ausdruckstyps (A für arithmetische Ausdrücke, B für boolesche Ausdrücke) verzichtet.*

Zuweisungen

Eine Zuweisung belegt eine Variable mit einem tatsächlichen Wert und ändert damit den Speicher. Die Belegung wird im Speicher σ gespeichert. Die Semantik von arithmetischen und booleschen Zuweisungen wird wie folgt definiert:

Definition 4.5 (Semantik von arithmetischer Zuweisungen)**.** Zuweisungen beginnen mit der Auswertung des Ausdrucks y der rechten Seite des Zuweisungsoperators :=. Dies wird notiert als $eval_A^\sigma(y)$. Darauf folgt die Speicherung des ermittelten Wertes val des Ausdrucks in den Speicher σ, notiert als $\sigma|_x^{val}$. Die Variable x steht dabei auf der linken Seite des Zuweisungsoperators, und $val \in INT$ und $x \in VAR$, wobei VAR die Menge der lokalen Programmvariablen enthält.

$$\text{Notation: } \sigma|_x^{val}(y) \hat{=} \begin{cases} val, & falls\, y = x \\ \sigma(y), & sonst \end{cases}$$

Die Notation aus Definition 4.5 wird auch für andere Funktionen verwendet.

Definition 4.6 (Semantik von boolescher Zuweisungen)**.** Zuweisungen beginnen mit der Auswertung des Ausdrucks y der rechten Seite des Zuweisungsoperators :=. Dies wird notiert als $eval_B^\sigma(y)$. Darauf folgt die Speicherung des ermittelten Wertes val des Ausdrucks in den Speicher σ, notiert als $\sigma|_x^{val}$. Die Variable x steht dabei auf der linken Seite des Zuweisungsoperators, und $val \in BOOL$ und $x \in VAR$.

4.2. Semantik der Beispielsprache $\mathcal{XYZ}$

Im nächsten Abschnitt wird die operationale Semantik, die die schrittweise Auswertung von Programmpunkt zu Programmpunkt eines beliebigen monolithischen Programms der Beispielsprache $\mathcal{XYZ}$ ausführt, vorgestellt. Zur Aufstellung der operationalen Semantik ist die Definition des Zustands notwendig. Um den Zustand eines Programms zu einem bestimmten Zeitpunkt formal zu beschreiben, wird eine Zustandsbeschreibung definiert.

Definition 4.7 (Zustand)**.** Der Zustand sei der prozess-algebraische Ausdruck $PEX(SFRAME)$ über die Menge $SFRAME$ (atomarer Prozesse) der Kellerelemente mit $SFRAME \subseteq NODE_\pi \times (VAR \to (INT \cup BOOL))$. Die Kellerelemente sind Paare (q, σ) aus einer endlichen Menge von Programmpunkten $NODE_\pi$ eines Programms π und dem Speicher σ mit $\sigma : VAR \to (INT \cup BOOL)$. Die Menge $PEX(SFRAME)$ ist die kleinste Menge, die folgende Bedingungen erfüllt:

- $SFRAME \subseteq PEX(SFRAME)$ und

- wenn (q, σ), $(q', \sigma') \in PEX(SFRAME)$, dann $(q, \sigma).(q', \sigma') \in PEX(SFRAME)$ und $(q, \sigma)||(q', \sigma') \in PEX(SFRAME)$,

wobei „||“ die parallele Komposition und „.“ die sequentielle Komposition kennzeichnet. Die parallele Komposition ist assoziativ und kommutativ. Die sequentielle Komposition ist assoziativ aber nicht kommutativ. Der Operator „.“ für die sequentielle Komposition bindet stärker als der parallele Operator „||“ für die parallele Komposition.

parallele Komposition „||“ und die sequentielle Komposition „.“ werden genutzt, um sequentielles und paralleles Verhalten von Prozedur- und Funktionsaufrufen darzustellen. Keller (engl. Stacks) können Rekursion modellieren, Petri-Netze können Nebenläufigkeit modellieren. Die Kaktuskeller-Semantik erlaubt, damit sowohl die Modellierung von Rekursion (sequentielles Verhalten) wie auch die Modellierung von Nebenläufigkeit (paralleles Verhalten).

Der Zustandsraum wird beschrieben durch $STATE = PEX(SFRAME)$. Dabei handelt es sich nicht um einen globalen Speicher im Sinne zustandsbehafteter Services.

Um die Semantik eines Programms darstellen zu können, wird im Folgenden auf Grundlage der Zustandsbeschreibung eine operationale Semantik definiert.

Definition 4.8 (Operationale Semantik für die Sprache $\mathcal{XYZ}$)**.** Die operationale Semantik der Beispielsprache $\mathcal{XYZ}$ wird als Auswertungsrelation $\rightarrow$, die den Zustand eines Programms am Programmpunkt q, mit dem Speicher σ angibt. Die Regeln der Auswertungsrelation $\rightarrow$ werden in den Abbildungen 4.1 bis 4.9 aufgelistet.

Die operationale Semantik beschreibt das mögliche Verhalten des Programms am entsprechenden Programmpunkt. Die Regeln beschreiben dabei das Verhalten von Anweisungen wie Deklarationen, bedingte Anweisungen und Verzweigungen, *return*-Anweisungen und Ausdrücke, wobei arithmetische und boolesche Ausdrücke erlaubt sind.

Bemerkung 4.3. *In den Abbildungen 4.1 bis 4.9 werden die Regeln der operationalen Semantik angezeigt und im Folgenden intuitiv erläutert. Die erste Spalte der Tabellen enthält den Namen der Regeln, die zweite Spalte die dazugehörige Inferenzregel und in der dritten Spalte die Anweisung oder Kontrollstruktur an dem entsprechenden Programmpunkt eines Programms.*

Die Regeln $MAIN_0$ und $MAIN_1$ stellen das Verhalten beim Aufruf der main-Methode eines Klienten dar. Wird kein Argument bei Aufruf (im Programmpunkt q_m) übergeben, so bleibt der Speicher leer (σ_0) vgl. Regel ($MAIN_0$) in Abbildung 4.1). Wird ein Argument übergeben, so wird dessen Wert im Speicher σ hinterlegt (vgl. Regel $MAIN_1$).

Die Semantik von Zuweisungen wird beschrieben durch die Regel (ZU) in Abbildung 4.2. Im Programmpunkt q wird mit der Zuweisung $x := e$; einer Programmvariable x

der Wert des Ausdrucks e zugewiesen. Die Wertzuweisung wird im Speicher σ hinterlegt. Der Programmpunkt q ist damit abgearbeitet und es wird der nächste Programmpunkt q' erreicht.

Die Regel $SKIP$ (Abb. 4.3) der operationalen Semantik beinhaltet die Anweisung *skip* (Abkürzung für $x := x$).

Die Semantik für bedingte Anweisungen $if-else$ werden durch die Regeln in Abbildung 4.4 definiert. Maßgeblich für den Folgezustand im Programmpunkt q ist die Auswertung des Ausdrucks e bezüglich des Speichers σ. Wird der Ausdruck mit *true* ausgewertet (IF_{true}) ist der Folgezustand der Programmpunkt q' und der Speicher σ wird in diesen Folgezustand übernommen und ändert sich nicht (IF_{true}). Wird der Ausdruck mit $false$ ausgewertet, so wird der *else*-Zweig ausgeführt und der Folgezustand ergibt sich aus dem Programmpunkt $\bar{q}$ und dem Speicher σ, der auch hier übernommen wird (IF_{false}). Das Verhalten des einseitigen if kann mit der Regel (IF_{false}) dargestellt werden. Die Anweisung s_2 enthält dabei eine *skip*-Anweisung und der Folgezustand ergibt sich aus dem Programmpunkt nach dem Rumpf des *else*-Zweigs, dem Programmpunkt $\bar{q}''$ und dem Speicher σ.

Die Regeln der operationalen Semantik in Abbildung 4.5 beschreiben das Verhalten beim Verlassen des if bzw. *else*-Rumpfs. Nach Ausführung des if- beziehungsweise *else*-Zweigs wird die Ausführung des Programms an dem Programmpunkt $\bar{q}''$ fortgeführt.

Nachdem die Regeln der operationalen Semantik für Zuweisungen und Kontrollstrukturen definiert wurden, anhand derer auch andere Konstrukte wie *break* oder *continue* im Schleifenrumpf erweitert werden können, werden nun die Regeln der Auswerterelation der operationalen Semantik zur Beschreibung des Verhaltens bei Prozeduraufrufen ohne Parameterübergabe in Abbildung 4.6 vorgestellt.

Beim Aufruf einer synchronen Prozedur (PS_0) an dem Programmpunkt q mit dem Speicher σ ergibt sich der neue Zustand aus dem Nachfolgeprogrammpunkt q' des Programmpunktes q mit dem Speicher σ sowie dem Programmpunkt q'', dem ersten Programmpunkt im Rumpf der aufgerufenen synchronen Prozedur p mit dem Speicher σ_0, ein leerer Speicher. Dieser leere Speicher wird mit Aufruf der asynchronen Prozedur p neu angelegt und bei Rückkehr wieder zerstört. Von besonderer Bedeutung ist hierbei die Benutzung der sequentiellen Komposition. Aus einem Kellerelement werden zwei Kellerelemente. Wobei nach den Inferenzregeln in Abbildung 4.10 nur das oberste Element (q'', σ_0) bearbeitet werden kann. Wenn die Ausführung das Ende des Rumpfs, die *return*-Anweisung im Programmpunkt q''', der aufgerufenen Prozedur erreicht, wird das oberste Kellerelement (q_m, σ') entfernt (pop). Das nächste Kellerelement (q', σ), dass den nächsten Programmpunkt q' nach dem Aufruf der synchronen Prozedur repräsentiert, kann abgearbeitet werden.

Der Unterschied der Regel (PS_0) zur Regel (PSR_0) liegt an der Rückgabe von z. Funk-

$(MAIN_0)$	$\overline{(q_m, \sigma_0) \to (q, \sigma_0)}$	$falls$ $q_m:$ $void\,main()\{$ $q:$ $s_1, \ldots$ $return;\}$
$(MAIN_1)$	$\overline{(q_m, \sigma_0) \to (q, \sigma\vert_x^{eval^{\sigma_0}(x)})}$	$falls$ $q_m:$ $void\,main(int\,x)\{$ $q:$ $s_1, \ldots$ $return;\}$

Wobei q_m, $q \in NODE$ und σ_0, σ Speicher sind.

Abbildung 4.1.: Regeln der Operationalen Kaktuskeller-Semantik für den Aufruf von main.

(ZU)	$\overline{(q, \sigma) \to (q', \sigma\vert_x^{eval_A^{\sigma}(e)})}$	$falls$ $q: x := e;$ $q': s_1, \ldots$

Wobei q, $q' \in NODE$ und σ ein Speicher ist.

Abbildung 4.2.: Regeln der Operationalen Kaktuskeller-Semantik für Zuweisungen.

(SKIP)	$\overline{(q, \sigma) \to (q', \sigma)}$	$falls$ $q: skip;$ $q': s_1; \ldots$

Wobei q, $q' \in NODE$ und σ ein Speicher ist.

Abbildung 4.3.: Regeln der Operationalen Kaktuskeller-Semantik für die Anweisung skip.

(IF_{true})	$\dfrac{eval_B^\sigma(e) = true}{(q,\sigma) \to (q',\sigma)}$	$falls$ $q:$ **if** $(e)\{$ $q':$ $s_1, \ldots$ $q'':$ $\}$ **else**$\{$ $\bar{q}:$ $s_2, \ldots$ $\bar{q}':$ $\}$ $\bar{q}'':$ $s_3, \ldots$
(IF_{false})	$\dfrac{eval_B^\sigma(e) = false}{(q,\sigma) \to (\bar{q},\sigma)}$	$falls$ $q:$ **if** $(e)\{$ $q':$ $s_1, \ldots$ $q'':$ $\}$ **else**$\{$ $\bar{q}:$ $s_2, \ldots$ $\bar{q}':$ $\}$ $\bar{q}'':$ $s_3, \ldots$

Wobei q, q', q'', $\bar{q}$, $\bar{q}'$, $\bar{q}'' \in NODE$ und σ ein Speicher ist.

Abbildung 4.4.: Regeln der Operationalen Kaktuskeller-Semantik für verzweigte Anweisungen.

(IF_{true_r})	$\overline{(q'',\sigma) \to (\bar{q}'',\sigma)}$	$falls$ $q:$ **if** $(e)\{$ $q':$ $s_1, \ldots$ $q'':$ $\}$ **else**$\{$ $\bar{q}:$ $s_2, \ldots$ $\bar{q}':$ $\}$ $\bar{q}'':$ $s_3, \ldots$
(IF_{false_r})	$\overline{(\bar{q}',\sigma) \to (\bar{q}'',\sigma)}$	$falls$ $q:$ **if** $(e)\{$ $q':$ s_1 $q'':$ $\}$ **else**$\{$ $\bar{q}:$ $s_2, \ldots$ $\bar{q}':$ $\}$ $\bar{q}'':$ $s_3, \ldots$

Wobei q, q', q'', $\bar{q}$, $\bar{q}'$, $\bar{q}'' \in NODE$ und σ ein Speicher ist.

Abbildung 4.5.: Regeln der Operationalen Kaktuskeller-Semantik für das Verlassen des if bzw. $else$-Rumpfs bei einer bedingten Anweisung.

tionsaufrufe verhalten sich wie Prozeduraufrufe, nur dass die Funktionsaufrufe in Ausdrücken $y := p()$; in Zuweisungen umgewandelt werden müssen. Bei Rückkehr nach q' muss der Wert von z im Speicher σ' in y im Speicher σ gespeichert werden. Hier wird der bei Rückkehr übergebene Speicher σ' bezüglich der Variable z ausgewertet und der Wert im Speicher σ in der y gespeichert.

Das Verhalten von Prozedur- und Funktionsaufrufen mit Parameterübergabe wird in Abbildung 4.7 mit den Regeln (PS_1) und (PSR_1) beschrieben. Dabei sind die Parameter lokale Variablen, die wie bei einer Zuweisung übergeben werden. Der Wert der Variablen x im Speicher σ wird in der Variable y im Speicher σ' gespeichert. Bei Rückkehr aus dem Funktionsrumpf wird das oberste Element entfernt und das nächste Kellerelement beschreibt den aktuellen Zustand. Das Programm befindet sich nach der Rückkehr bei der Ausführung im Programmpunkt q' mit Speicher σ. Die Parameterübergabe bei Funktionsaufrufen in (PSR_1) funktioniert wie in der Regel (PS_1) beschrieben. Die Rückgabe verläuft analog zu (PSR_0).

Bis zu diesem Zeitpunkt ist das beschriebene Verhalten der Beispielsprache $\mathcal{XYZ}$ das Verhalten einer imperativen sequentiellen Programmiersprache, die zum Beispiel um Ausnahmebehandlung [29] erweitert werden könnte. Neben synchronen Prozeduren und Funktionen unterstützt die Beispielsprache $\mathcal{XYZ}$ auch die Möglichkeit, Prozeduren und Funktionen asynchron aufzurufen. Beim Aufruf einer asynchronen Prozedur oder Funktion wird parallel verzweigt, sodass der Aufrufer und die aufgerufene Prozedur parallel arbeiten können. In der Definition des Zustandes wurde bereits der „$\|$ “-Operator eingeführt. Um die parallele Verzweigung zu kennzeichnen, wird dieser Operator genutzt. Die Parameterübergabe erfolgt wie bei synchronen Prozeduren und Funktionen. Bei der Rückkehr von Funktionen ist eine Wertübergabe nur über eine explizite Synchronisationsanweisung *sync* möglich.

Zur Darstellung des Verhaltens von asynchronen Prozedur- und Funktionsaufrufen wird der Typ $FUTURE$ eingeführt. Neben dem neuen Typ $FUTURE$ wird ein globaler Zähler Φ eingeführt, der eine eindeutige Kennung von asynchronen Prozedur- und Funktionsaufrufen ermöglicht.

Definition 4.9 (Future)**.** Der Typ $FUTURE \subset VALUE$ ist die Menge von Paaren $(p, i) \in (PROC \times \mathbb{N})$, wobei i durch den globalen Zähler Φ belegt wird. Elemente $(p, i) \in FUTURE$ heißen future. Φ wird bei jedem asynchronen Prozedur- oder Funktionsruf aufgerufen und inkrementiert. p sei der Name der aufgerufenen Prozedur oder Funktion, mit $p \in PROC$ gilt und $PROC$ die Menge der Prozeduren und Funktionen eines Programms ist. Eine Programmvariable vom Typ $FUTURE$ ist eine **single assignment** (dt. einzige Zuweisung) Variable. An diese Programmvariable kann nur einmal zugewiesen werden.

Bemerkung 4.4. *Auch in Schleifen können futures benutzt werden. Um sicherzustellen, dass auch hier Variablen vom Typ FUTURE lokale Variablen sind und damit nur einmal zugewiesen werden könnnen, wird jede Schleife in eine tail-recursion function umge-*

(PS_0)	$\dfrac{\overline{(q,\sigma) \to (q'',\sigma_0).(q',\sigma)}}{\overline{(q''',\sigma').(q',\sigma) \to (q',\sigma)}}$,	$falls$ $q:$ $p()$ $q':$ $s_1, \ldots$ $void\, p()\{$ $q'':$ $\quad s_2, \ldots$ $q''':$ $\quad return;\}$
(PSR_0)	$\dfrac{\overline{(q,\sigma) \to (q'',\sigma_0).(q',\sigma)}}{(q''',\sigma').(q',\sigma) \to (q',\sigma\vert_y^{eval_A^{\sigma'}(z)})}$	$falls$ $q:$ $y := p()$ $q':$ $s_1, \ldots$ $int\, p()\{$ $q'':$ $\quad s_2, \ldots$ $q''':$ $\quad return\, z;\}$

Wobei q, q', q'', $q''' \in NODE$ und σ, σ_0, σ' Speicher sind.

Abbildung 4.6.: Regeln der Operationalen Kaktuskeller-Semantik für synchrone Prozedur- und Funktionsaufrufe ohne Parameterübergabe.

(PS_1)	$\dfrac{\overline{(q,\sigma) \to (q'',\sigma\vert_y^{eval^{\sigma}(x)}).(q',\sigma)}}{\overline{(q''',\sigma').(q',\sigma) \to (q',\sigma)}}$,	$falls$ $q:$ $p(x)$ $q':$ s_1 $void\, p(int\, y)\{$ $q'':$ $\quad s_2, \ldots$ $q''':$ $\quad return;\}$
(PSR_1)	$\dfrac{\overline{(q,\sigma) \to (q'',\sigma\vert_y^{eval^{\sigma}(x)}).(q',\sigma)}}{(q''',\sigma').(q',\sigma) \to (q',\sigma\vert_y^{eval^{\sigma'}(z)})}$	$falls$ $q:$ $y := p(x)$ $q':$ s_1 $int\, p(int\, y)\{$ $q'':$ $\quad s_2, \ldots$ $q''':$ $\quad return\, z;\}$

Wobei q, q', q'', $q''' \in NODE$ und σ, σ' Speicher sind.

Abbildung 4.7.: Regeln der Operationalen Kaktuskeller-Semantik für synchrone Funktionsaufrufe mit Parameterübergabe.

(PA_0)	$\dfrac{\overline{(q,\sigma) \to (q', \sigma'|_f^{(p,\Phi)})||(\bar{q}, \sigma''|_p^{\Phi})}, \quad \sigma(f) = (p, \sigma'(p))}{(q'', \sigma)||(\bar{q}', \sigma') \to (q'', \sigma)}$	$falls$ $\ldots$ $q^f:$ $futuref;$ $\ldots$ $q:$ $f := p()$ $q':$ $s_1, ...$ $q'':$ $return;\}$ $void\|int\, p()\{$ $\bar{q}:$ $s_2, \ldots$ $\bar{q}':$ $return;\}$
(PA_1)	$\dfrac{\overline{(q,\sigma) \to (q', \sigma|_f^{(p,\Phi)})||(\bar{q}, \sigma'|_{y,p}^{eval^{\sigma}(v),\Phi})}, \quad \sigma(f) = (p, \sigma'(p))}{(q'', \sigma)||(\bar{q}', \sigma') \to (q'', \sigma)}$	$falls$ $\ldots$ $q^f:$ $futuref;$ $q:$ $f := p(v)$ $q':$ $s_1, ...$ $q'':$ $return;\}$ $void\|int\, p(int\, y)\{$ $\bar{q}:$ $s_2, \ldots$ $\bar{q}':$ $return;\}$

Wobei q, q', q'', $\bar{q}$, $\bar{q}' \in NODE$ und σ, σ' Speicher sind und Φ globaler Zähler für asynchrone Aufrufe ist.

Abbildung 4.8.: Regeln der Operationalen Kaktuskeller-Semantik für asynchrone Prozeduraufrufe mit und ohne Parameterübergabe sowie mit und ohne Rückgabewert.

wandelt. Dadurch wird sichergestellt, dass keine schleifenübergreifende futures existieren können und die futures lokale Schleifenvariablen sind. Die Synchronisation kann dann nur in derselben Iteration ausgeführt werden.

Wie in Regel (PA_0) zu sehen ist, wird mit Aufruf dem obersten Kellerelement, das den Nachfolgeprogrammpunkt und -speicher symbolisiert, mit Hilfe des Symbols $||$, mit dem Kellerelement, dass den asynchronen Aufruf der Prozedur p symbolisiert, zu einem prozess-algebraischen Ausdruck $(q', \sigma'|_f^{(p,\Phi)})$ hinzugefügt. Die Synchronisation wird durch die jeweils zweite Regel in (PA_0) und (PA_1) beschrieben. Um diese Regeln ausführen zu können, müssen die Bedingungen $\sigma(f) = (p, \sigma'(p))$ erfüllt sein. Die zu synchronisierende Prozedur oder Funktion p muss von der Prozedur oder Funktion, die synchronisieren möchte, die asynchrone Prozedur oder Funktion aufgerufen haben. Wird der Ausdruck erfüllt, so wird das Kellerelement, was den Programmpunkt an der *return*-Anweisung der asynchronen Prozedur beschreibt, mit dem Kellerelement der *return*-Anweisung der aufrufenden Prozedur synchronisiert. Dabei wird der Speicher σ der aufrufenden Prozedur behalten.

Da Rückgabewerte von asynchronen Funktionsaufrufen nur über explizite Synchronisationsanweisungen *sync* abgefangen werden können, gelten die gleichen Regeln für die Synchronisation mit der *return* Anweisung bei Funktionsaufrufen (Regel (PA_1)).

Die Regeln (PA_{sync}) und (PAR_{sync}) der Operationalen Kaktuskeller-Semantik in Abbildung 4.9 beschreiben das Verhalten des Synchronisationsmechanismus *sync* mit und ohne Rückgabewert. Bei beiden Regeln werden zwei parallele Kellerelemente, die durch den parallelen Operator „$\|$“ miteinander verknüpft sind, zu einem Kellerelement synchronisiert. Vor Ausführung der Regel muss die Prämisse erfüllt sein. Die Prozedur oder Funktion aus der die Prozedur p synchronisiert werden soll, muss die Prozedur p vorher aufgerufen haben, vgl. mit Programmpunkt q in den Regeln (PA_{sync}) und (PAR_{sync}). Ist die Prämisse, dass im Speicher σ unter dem future f die Prozedur p und der $i-te$ Aufruf, und im Speicher σ' unter der Variable p der Wert i gespeichert ist, kann nicht synchronisiert werden. Die zu synchronisierende Prozedur oder Funktion wurde nicht aus der Prozedur bzw. Funktion in der das *sync f* ausgeführt werden soll, aufgerufen (p ist nicht an f in σ gebunden). Oder die zu synchronisierende Prozedur p, wird nicht von der Prozedur synchronisiert, die sie aufgerufen hat (p ist nicht an f in σ' gebunden). Ist die Prämisse erfüllt, so wird die Regel angewandt. Der Programmpunkt q'' mit dem Speicher σ und der Programmpunkt $\bar{q}'$ (*return*-Anweisung der Prozedur p) wird synchronisiert, in dem der Keller, der für den asynchronen Aufruf von p erzeugt wurde und in dem nur noch das Kellerelement mit dem Programmpunkt $\bar{q}'$ liegt, beseitigt wird. Gibt die asynchron aufgerufene Funktionen einen Rückgabewert zurück, so wird dieser im Speicher σ gespeichert.

Bemerkung 4.5. *Der Zustand einer nicht prototypischen Programmiersprache enthält mehr Informationen, dazu gehören zum Beispiel Informationen zum Gültigkeitsbereich bzw. eine Umgebung $\rho : VAR \to ADDR$. Außerdem können typische Konzepte aus*

(PA_{sync})	$\dfrac{\sigma(f) = (p, \sigma'(p))}{(\bar{q}', \sigma')\|\|(q'', \sigma) \to (q''', \sigma)}$	$falls$ $\ldots$ $q^f:$ $futuref;$ $\ldots$ $q:$ $f := p(x)$ $q':$ $s_1, \ldots$ $q'':$ $sync\, f;$ $q''':$ $s_2, \ldots$ $q'''':$ $return\, z;$ $void\, p(int\, y)\{$ $\bar{q}:$ $\quad s_3$ $\bar{q}':$ $return;\}$
(PAR_{sync})	$\dfrac{\sigma(f) = (p, \sigma'(p))}{(q'', \sigma)\|\|(\bar{q}', \sigma') \to (q''', \sigma\|_x^{eval^{\sigma'}(y)})}$	$falls$ $\ldots$ $q^f:$ $futuref;$ $\ldots$ $q:$ $f := p(x)$ $q':$ $s_1, \ldots$ $q'':$ $x := sync\, f;$ $q''':$ $s_2, \ldots.$ $q'''':$ $return;$ $\ldots$ $int\, p(int\, y)\{$ $\bar{q}:$ $s_3, \ldots$ $\bar{q}':$ $return\, y;\}$

Wobei q, q', q'', q''', q'''', $\bar{q}, \bar{q'} \in NODE$ und σ, σ' Speicher sind.

Abbildung 4.9.: Regeln der Operationalen Kaktuskeller-Semantik für Synchronisationsbarrieren ohne Rückgabewert (PA_{sync}) und mit Rückgabewert (PAR_{sync}).

der Semantik verwendet werden, z.B. Continuations und Ausnahmebehandlung. Fallunterscheidungen, Polymorphie und Typfallunterscheidung sind Verallgemeinerungen von Verzweigungen. Für die Deadlockanalyse, die in dieser Arbeit betrachtet wird, spielen diese Erweiterungen keine Rolle, da später vom Speicher abstrahiert wird.

Definition 4.10 (Semantik eines Programms)**.** Die Semantik eine Programms Π der Beispielsprache $\mathcal{XYZ}$ wird durch ein Zustandsübergangssystem $[\![\Pi]\!]$ definiert.

Definition 4.11 (Zustandsübergangssystem)**.** Ein Zustandsübergangssystem wird definiert durch ein Quadrupel $oS = (Z, \Rightarrow_{oS}, q, F)$, dabei sei:

- $Z \subseteq PEX(SFRAME)$ eine Menge von Ausdrücken über Kaktuskeller-Elemente,
- $q \hat{=} (q_m, \sigma_0) \in SFRAME$ der Startzustand mit leeren Speicher σ_0,
- F ist die einelementige Menge mit dem Element $(q_f, \sigma_f) \in SFRAME$ der Finalzustand,
- $\Rightarrow_{oS} \subseteq PEX(SFRAME) \times PEX(SFRAME)$ eine Zustandsübergangsrelation nach Abb. 4.1 bis 4.9.

Der Startzustand (q_m, σ_0) entspricht dem Programmpunkt der ersten Anweisung im Rumpf von main. Die einelementige Menge F mit dem Element $(q_f, \sigma_f) \in SFRAME$ entspricht dem Erreichen der *return*-Anweisung von main.

Die schrittweise Ausführung eines Programms wird durch die Anwendung der Regeln der operationale Semantik in Abbildung 4.1 bis 4.9 vorgegeben.

Bemerkung 4.6. *Anhand der Definition 4.11 ist zu erkennen, dass im Vergleich mit der Definition der Process Rewrite System von [39] Ähnlichkeiten bestehen. Es existiert ein Startzustand, ein Finalzustand, ein Regelsystem ($\Rightarrow_{oS}$) und eine Menge an prozessalgebraischen Ausdrücken. Der Unterschied zwischen dem PRS und der operationalen Semantik oS besteht darin, dass die Menge der $SFRAME$ nicht endlich ist.*

Die definierte Ableitungsrelation $\Rightarrow_{oS} \subseteq PEX(SFRAME) \times PEX(SFRAME)$ genügen den Inferenzregeln in Abbildung 4.10. Die Inferenzregeln definieren Anwendung der Ableitungsrelation zur Darstellung der schrittweisen Ausführung eines Programms.

Definition 4.12 (Lauf)**.** Ein Lauf ist die Ausführung eines Zustandsübergangssystems oS, eine Sequenz $e_0, e_1, e_2, ...e_n$ von prozess-algebraischen Ausdrücken $PEX(SFRAME)$, wobei e_0 dem Anfangszustand mit leeren Speicher entspricht, so dass $e_i \Rightarrow_{oS} e_{i+1}$, $i = 0, ..., n-1$ und $n, i \in \mathbb{N}$ mit $e_i \Rightarrow_{oS} e_{i+1}$ bewiesen werden kann, ohne die Regel (T) der Inferenzregeln in Abbildung 4.10 zu nutzen.

$$\frac{(q,\sigma) \to (q',\sigma')}{(q,\sigma) \Rightarrow (q',\sigma')} \quad \text{(L)}$$

$$\frac{(q,\sigma) \Rightarrow (q',\sigma')}{(q,\sigma).(q'',\sigma'') \Rightarrow (q',\sigma').(q'',\sigma'')} \quad \text{(S)}$$

$$\frac{(q,\sigma) \Rightarrow (q',\sigma') \quad (q',\sigma') \Rightarrow (q'',\sigma'')}{(q,\sigma) \Rightarrow (q'',\sigma'')} \quad \text{(T)}$$

$$\frac{(q,\sigma) \Rightarrow (q',\sigma')}{(q,\sigma) \parallel (q'',\sigma'') \Rightarrow (q',\sigma') \parallel (q'',\sigma'')} \quad \text{(P1)}$$

$$\frac{(q,\sigma) \Rightarrow (q',\sigma')}{(q'',\sigma'') \parallel (q,\sigma) \Rightarrow (q'',\sigma'') \parallel (q',\sigma')} \quad \text{(P2)}$$

$$\frac{}{(q,\sigma) \Rightarrow (q,\sigma)} \quad \text{(R)}$$

Abbildung 4.10.: Inferenzregeln über die Ableitungsrelationen von Zustandsübergangssystemen.

Die Ableitungsrelation ist die reflexive transitive Hülle $\Rightarrow^*_{oS}$ der direkten Ableitungsrelation $\Rightarrow_{oS}$.

Intuitiv bedeutet das, dass immer nur exakt eine Regel der Ableitungsrelation des oS angewandt werden kann und dass die Ableitungssequenz $e_0, ..., e_n$ die schrittweise Ausführung der operationalen Semantik des Programms darstellt.

Definition 4.13 (Normalform)**.** Sei oS ein Zustandsübergangssystem mit direkter Ableitungsrelation $\Rightarrow_{oS}$. Ein prozess-algebraischer Ausdruck e mit $e \in PEX(SFRAME)$ heißt Normalform gdw. es keinen prozess-algebraischen Ausdruck $e' \in PEX(SFRAME)$ mit $e \neq e'$ und $e \Rightarrow_{oS} e'$ gibt.

Definition 4.14 (Deadlock)**.** Sei oS ein Zustandsübergangssystem und d ein prozess-algebraischer Ausdruck mit $d \in PEX(SFRAME)$. Der prozess-algebraische Ausdruck d ist erreichbar gdw. $e_0 \Rightarrow d$, wobei e_0 dem Anfangszustand (q_0, σ_0) entspricht. Ein erreichbarer prozess-algebraischer Ausdruck $d \in PEX(SFRAME) \setminus F$ ist ein Deadlock gdw. kein Zustand $e' \in PEX(SFRAME)$ existiert mit $e' \in PEX(SFRAME)$ und $d \neq e'$, so dass $d \Rightarrow_{oS} e'$ist.

Intuitiv bedeutet das, dass auf eine Deadlocksituation keine Regel der operationalen Semantik angewandt werden können und die erhaltene Situation nicht dem Finalzustand (q_f, σ_f) des Zustandsübergangssystem oS entspricht.

Graphische Darstellung

Zur besseren Verständlichkeit kann das Verhalten eines Programms nicht nur als Ableitungsrelation des Zustandsübergangssystem durch Anwendung der Regeln der operationalen Semantik unter Beachtung der Inferenzregeln gezeigt werden, sondern auch durch einen Kaktuskeller grafisch dargestellt werden.

Kaktuskeller wurden bereits eingeführt als Bäume von Kellern von [19], vgl. Def. 3.2.3. Das in dieser Arbeit vorgestellte Ausführungsmodell erlaubt unbeschränkte Rekursion und unbeschränkte Nebenläufigkeit. Diese Zustände können mit einem Kaktuskeller, der um einen Speicher erweitert wird, dargestellt werden. Das Laufzeitsystem vereinigt Petri-Netze und Laufzeitkeller zu einem Kaktuskeller. Jeder Prozess repräsentiert dabei einen Keller mit entsprechenden Speicher. Auf den obersten Elementen der einzelnen Keller des Kaktuskellers können Regeln angewendet werden. Wenn eine Anweisung ausgeführt wird, dann wird das oberste Kellerelement vom Keller entfernt und durch ein neues mit aktuellen Programmpunkt und aktualisierten Speicher ersetzt. Wenn eine synchrone Prozedur aufgerufen wird, dann wird entsprechend ein neues Kellerelement auf den Keller gelegt, welches diesen neuen Prozess symbolisiert. Wird eine asynchrone Prozedur aufgerufen, wird ein neuer Keller erzeugt. Das oberste Kellerelement des aufrufenden Prozesses und das unterste Kellerelement des neuen Kellers sind miteinander verlinkt, und damit ähnlich aufgebaut wie ein Saguaro Kaktus (Abb. B.1 im Anhang B). Eine Synchronisation kann nur zwischen den obersten Kaktuskeller-Elementen ausgeführt werden. Bei einer Synchronisation wird der Keller, der den asynchronen Aufruf repräsentiert, eliminiert. Ein Deadlock in einem Kaktuskeller bedeutet eine lokaler Deadlock auf jedem Keller des Kaktuskellers. Es ist also keine Regel auf keinem Kaktuskeller mehr anwendbar.

In Tabelle 4.1 ist ein Auszug einzelner Regeln der operationalen Semantik und der dazugehörigen grafischen Kaktuskeller-Darstellung gezeigt. Speicher werden dabei folgendermaßen dargestellt: $\sigma \triangleq [x_1 := v_1, \ldots, x_n := v_n]$ ist die Funktion $\sigma(x_i) = v_i, i = 1, \ldots, n$ und $\sigma(y) = \perp$ für alle $y \notin \{x_1, \ldots, x_n\}$. Ansonsten wird die im Grundlagenkapitel 3.2.3 gezeigte grafische Darstellung der regulären Kaktuskeller (Tab. 3.1) verwendet.

Beispiel 4.2. Um die Anwendung der operationalen Kaktuskeller-Semantik zu zeigen, wird das Service-orientierte System in Abbildung 4.11 vorgestellt. Es besteht aus drei Services, wobei der Service S den Klienten darstellt, der über die Nutzungsschnittstelle R_S den Service A mit der Funktion a aufruft, die in der benötigten Schnittstelle R_S (Angebotsschnittstelle) beschrieben wird. Service A besitzt neben der angebotenen Schnittstelle I_A die Nutzungsschnittstelle R_A. Der Service A implementiert die asynchrone Prozedur b nicht selbst, sondern benutzt die Implementierung von Service B über die Angebotsschnittstelle I_B (vgl. Schnittstellenbeschreibung I_B).

Um die operationale Kaktuskeller-Semantik nutzen zu können, wird vorausgesetzt, dass

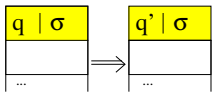
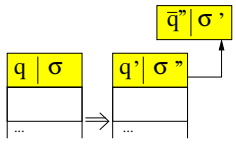
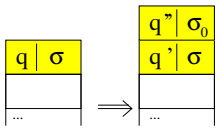
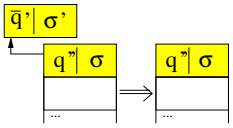
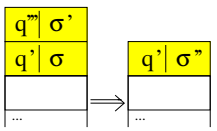
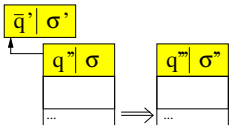
Regel der Operationalen Semantik	Kaktuskeller	Regel der Operationalen Semantik	Kaktuskeller
(ZU) Zuweisung mit $\sigma = \sigma\|_x^{eval^\sigma(e)}$		(PA_1) asynchroner Funktionsaufruf mit Parameterübergabe $\sigma'' = \sigma\|_f^{(p,\Phi)}$ $\sigma' = \sigma_0\|_{y,p}^{eval^\sigma(v),\Phi}$	
(PS_0) synchroner Prozeduraufruf		(PA_1) return des asynchronen Funktionsaufrufs mit Wertrückgabe	
(PSR_0) return eines synchronen Funktionsaufrufs $\sigma'' = \sigma\|_y^{eval^{\sigma'}(z)}$		(PAR_{sync}) Synchronisierung $\sigma'' = \sigma\|_x^{eval^{\sigma'}(y)}$	

Tabelle 4.1.: Grafische Darstellung der operationalen Kaktuskeller-Semantik bei Zuweisungen, synchronen und asynchronen Prozedur- und Funktionsaufrufen.

es sich bei dem vorliegenden Service-orientierten System um White-Boxes handelt. Sämtlicher Quellcode ist vorhanden und damit die Eingangs- und Ausgangspunkte der Prozeduren und Funktionen der jeweiligen Schnittstellen.

Das Verhalten für das in Abbildung 4.11 gezeigte Service-orientierte System wird durch das Zustandsübergangssystem $SAB = (Z, \Rightarrow_{SAB}, q, F)$ definiert, dabei sei:

- $Z = \{(q_m, \sigma_0), (q_0, \sigma_1), (q_1, \sigma_2), (i_a, \sigma_3).(q_f, \sigma_2), (q_{a1}, \sigma_4).(q_f, \sigma_2),$
  $(re_a, \sigma_5) \parallel (i_b, \sigma_6).(q_f, \sigma_2), (re_a, \sigma_5) \parallel (re_b, \sigma_7).(q_f, \sigma_2), (re_a, \sigma_5).(q_f, \sigma_2),$
  $(q_f, \sigma_8)\}$,
- $\Rightarrow_{SAB}$ dargestellt als Lauf in Abbildung 4.12,
- $q \hat{=} (q_m, \sigma_0)$,
- $F \hat{=} \{(q_f, \sigma_8)\}$.

Die Anwendung der Regeln der operationalen Kaktuskeller-Semantik ist in Abbildung 4.12 zu sehen. Die erste Spalte gibt an, welcher prozess-algebraische Ausdruck dem jeweiligen Kaktuskeller in der grafischen Darstellung in Tabelle 4.2 entspricht, in der zweit-

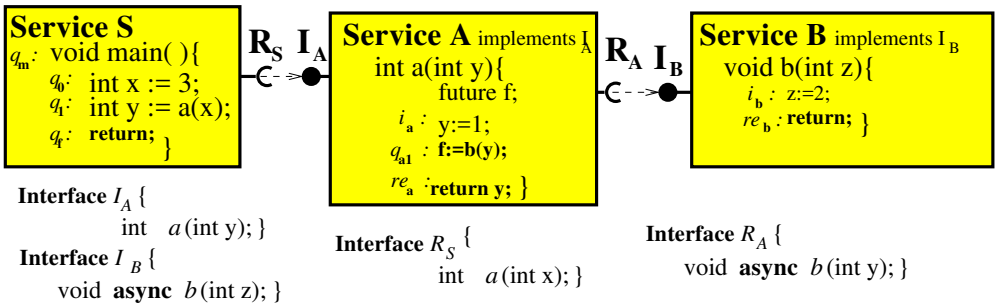


Abbildung 4.11.: Ein Service-orientiertes Softwaresystem SAB mit den Services S, A und B und den dazugehörigen Schnittstellen R_S, I_A, R_A und I_B.

en Spalte wird der aktuelle Zustand als prozess-algebraischer Ausdruck gezeigt und in der dritten Spalte, welche Regel angewandt wurde. Der unterstrichene Teil des prozess-algebraischen Ausdrucks stellt dabei den Teil des prozess-algebraischen Ausdrucks dar, auf den die jeweilige Regel aus Spalte zwei angewandt wird. Das Service-orientierte System aus Abbildung 4.11 kommt bei Ausführung vom Startzustand (q_m, σ_0) in den Finalzustand (q_f, σ_8) wobei der Speicher σ_8 den Speicherinhalt $[x := 3, y = 1]$ besitzt.

Kellerkaktus in Tab. 4.2	Lauf	Regel aus $\Rightarrow_{SAB}$
(1)	$\underline{(q_m, \sigma_0)} \Rightarrow$	$(MAIN)$
(2)	$\underline{(q_0, \sigma_1)} \Rightarrow$	(ZU)
(3)	$\underline{(q_1, \sigma_2)} \Rightarrow$	(PSR_1)
(4)	$\underline{(i_a, \sigma_3)}.(q_f, \sigma_2) \Rightarrow$	(ZU)
(5)	$\underline{(q_{a1}, \sigma_4)}.(q_f, \sigma_2) \Rightarrow$	(PA_1)
(6)	$(re_a, \sigma_5) \parallel \underline{(i_b, \sigma_6)}.(q_f, \sigma_2) \Rightarrow$	(ZU)
(7)	$(re_a, \sigma_5) \parallel \underline{(re_b, \sigma_7)}.(q_f, \sigma_2) \Rightarrow$	(PA_1)
(8)	$\underline{(re_a, \sigma_5).(q_f, \sigma_2)} \Rightarrow$	(PSR_1)
(9)	$\underline{(q_f, \sigma_8)}$	*Finalzustand*

Abbildung 4.12.: Lauf der operationalen Semantik auf das Service-orientierte System in Abbildung 4.11

In Tabelle 4.2 ist der Lauf des Service-orientierten Systems von Abbildung 4.11 grafisch dargestellt. Die obersten Kellerelemente sind gelb eingefärbt. Auf diese Elemente können die Regeln der operationalen Semantik angewandt werden. Der synchrone Aufruf der Funktion a wird dargestellt durch den Übergang des Kaktuskellers (3) in den Kaktuskeller (4) in der Tabelle 4.2 durch Anwendung der Regel (PSR_1). Auf das oberste Kellerelement wird das Kellerelement (i_a, σ_3) gelegt, wobei i_a der Eintrittprogrammpunkt der Funktion a von Service A und σ_3 der lokale Speicher von Service A mit dem Inhalt $[y := 3]$ ist. Der Speicher beinhaltet $y := 3$, da der übergebene Wert x aus Service M im lokalen Speicher σ_2 von M den Wert 3 hat.

Beim Aufruf der asynchronen Methode b von Service B aus dem Programmpunkt q_{a1}

Speicher	Inhalt
σ_0	[]
σ_1	$[x := 1]$
σ_2	$[x := 3]$
σ_3	$[y := 3]$
σ_4	$[y := 1]$
σ_5	$[y := 1, f := (b, 1)]$
σ_6	$[z := 1, b := 1]$
σ_7	$[z := 2, b := 1]$
σ_8	$[x := 3, y := 1]$

Abbildung 4.13.: Der Inhalt der Speicher aus Abbildung 4.11.

(vgl. Keller (5)) wird ein neuer Keller (vgl. Keller (6)) erzeugt, der als Kellerelement den Eintrittsprogrammpunkt der asynchronen Methode b sowie den lokalen Speicher σ_6 enthält, (vgl. (i_b, σ_6)). Zusätzlich zur Übergabe des Parameters y von Service A wird außerdem in der Variable p die Anzahl der Aufrufe asynchroner Funktionen, die getätigt wurden, gespeichert. Das $future$ mit der Information über die aufgerufene asynchrone Prozedur sowie die Anzahl der Aufrufe asynchroner Prozeduren Φ ist im Speicher des aufrufenden Kellerelementes (re_a, σ_5) mit dem Speicherinhalt $[y := 1, f := (b, 1)]$ gespeichert. Das unterste Kellerelement des neuen Keller bleibt bis zum Abbau mit dem aufrufenden Kellerelement (re_a, σ_5) verbunden. Auf dieses Kellerelement könnten, wenn es weitere Anweisungen nach dem aufrufenden Punkt q_{a1} gäbe, weitere Regeln angewandt werden. Nach dem vorliegenden Ausführungsmodell kann der neue Keller nur abgebaut werden, wenn das mit ihm verbundene Kellerelement die *return*-Anweisung oder ein *sync* erreicht und in den Speichern σ_5 das future mit der Prozedur und der Anzahl der Aufrufe asynchroner Funktionen vorhanden ist und im Speicher σ_7 unter der Variable b die Anzahl der Aufrufe gebunden ist. In diesem Fall, wird die *return*-Anweisung der Funktion a erreicht, (vgl. (re_a, σ_5)) mit dem Speicherinhalt $[y := 1, f := (b, 1)]$.

Bei dem in diesem Unterkapitel präsentierten Service-orientierten System wurde davon ausgegangen, dass es sich um $Whitebox$-Services handelt, die bereits über bestimmte Schnittstellen fest miteinander verbunden sind. Der Fokus der Arbeit liegt auf der Deadlockanalyse von Service-orientierten Systemen. Da ein großer Vorteil Service-orientierter Systeme darin besteht, dass eine Bindung von beliebigen Services mit der gleichen Schnittstellenimplementierung beziehungsweise -beschreibung möglich ist, soll die operationale Semantik so erweitert werden, dass auch hier das Verhalten dargestellt werden kann. Wie die Anwendung der operationalen Kaktuskeller-Semantik auf Service-orientierte Softwaresysteme mit dieser Eigenschaft ermöglicht wird, erläutert das nächste Unterkapitel.

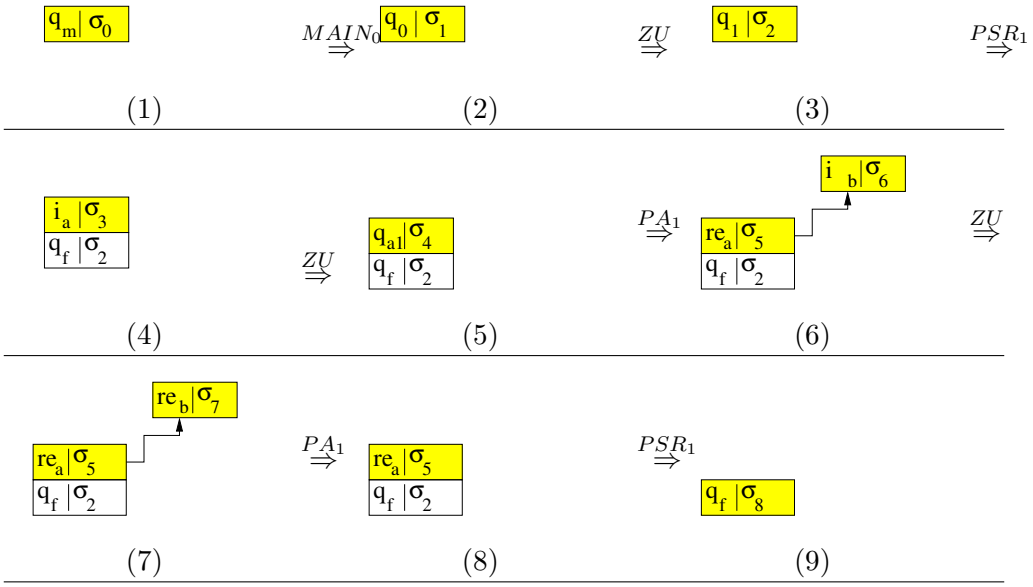


Tabelle 4.2.: Anwendung der Regeln der operationalen Kaktuskeller-Semantik auf das Service-orientierte System aus Abbildung 4.11

4.3. Semantik Service-orientierter Systeme

Im Folgenden wird die operationale Kaktuskeller-Semantik aus dem Kapitel 4.2 erweitert, um das Verhalten von Services mit benötigten und angebotenen Schnittstellen, darstellen zu können. So kann das Verhalten der einzelnen Services vor der Komposition bestimmt und zur Verfügung gestellt werden.

Dafür wird im ersten Teil die Erweiterung der operationalen Semantik und die Komposition des Verhaltens der Services definiert. Im zweiten Teil wird das Vorgehen anhand eines Beispiels erläutert. Im abschließenden Teil wird das Verfahren genutzt, um das Verhalten eines Service-orientierten Systems an einem konkreten Beispiel zu zeigen.

4.3.1. Semantik von Services

Um das Verhalten eines Service-orientierten Systems zu erfassen, wird das Verhalten der Services, die das Service-orientierte System bilden, benötigt. Dazu wird die operationale Semantik aus Def. 4.8 der Beispielsprache $\mathcal{XYZ}$ auf die Services angewandt. Für Prozedur- oder Funktionsaufrufe über Servicegrenzen hinaus (Nutzung der Schnittstellen)

werden Platzhalter benötigt, da konkrete Eintritts- und Rückkehrprogrammpunkte der Prozeduren und Funktionen, die über angebotene Schnittstellen zur Verfügung gestellt werden, nicht bekannt sind. Diese Dummy-Einstiegs- und Rückkehrpunkte werden erst bei Bindung des Services mit einem anderen Service über ihre Schnittstellen bekannt. Diese Platzhalter bestehen aus einem Dummy-Programmpunkt und einem Dummy-Speicher. Außerdem erhält jeder Service mit einer angebotenen Schnittstelle je Prozedur beziehungsweise Funktion zwei zusätzliche Programmpunkte. Diese zusätzlichen Programmpunkte sind die neuen Einstiegs- und Rückkehrpunkte aller Prozeduren und Funktionen, die über die Angebotsschnittstellen zur Verfügung gestellt werden. Die zusätzlichen Programmpunkte werden benötigt, um den Inhalt der Dummy-Speicher auf die regulären Speicher zu übertragen. Neben den Platzhaltern und den zusätzlichen Programmpunkten wird eine Platzhalterfunktion benötigt. Diese Platzhalterfunktion wird vom Service, der die angebotene Prozedur oder Funktion implementiert (Angebotsschnittstelle), zur Verfügung gestellt. Bei der Komposition der Services werden mit Hilfe der Platzhalterfunktion, die jeweiligen Platzhalter auf die zusätzlichen Programmpunkte abgebildet.

Definition 4.15 (Platzhalter - DFRAME)**.** Sei $DFRAME_S$ die Menge der Platzhalter eines Service S mit

$$DFRAME_S \subseteq DNODE_S \times [VAR \rightarrow INT \cup BOOL \cup FUTURE].$$

$DFRAME$ ist die Menge der Tupel (d, σ^{dy}), wobei $d \in DNODE_S$ ein Dummy-Programmpunkt und σ^{dy} ein Dummy-Speicher von Service S sei.

Definition 4.16 (Zusätzliche Kaktuskeller-Elemente - NFRAME)**.** Sei $NFRAME_S$ die Menge der zusätzlichen Kaktuskeller eines Service S mit

$$NFRAME_S \subseteq NNODE_S \times [VAR \rightarrow INT \cup BOOL \cup FUTURE].$$

$NFRAME$ ist die Menge der Tupel (z, σ^{dy}), wobei $z \in NNODE_S$ ein zusätzlicher Programmpunkt ist und σ^{dy} ein Dummy-Speicher sei.

Definition 4.17 (Dummy-Programmpunkt - $DNODE_S$)**.** Sei $DNODE_S$ die Menge der Dummy-Programmpunkte eines beliebigen Service S, wobei $DNODE_S \cap NODE_S = \emptyset$. Die Menge der $DNODE_S$ ergibt sich aus der Vereinigung der Menge der benötigten Dummy-Programmpunkte $DNODE^{R_S}$ und der angebotenen Dummy-Programmpunkte $DNODE^{I_S}$.

Für alle Prozeduren oder Funktionen $p \in R_S$ (in der Nutzungsschnittstelle) eines Service S existiert ein:

- $init_p^{R_S} \in DNODE^{R_S}$, ein Dummy-Programmpunkt, der den Aufruf einer Prozedur p in der benötigten Schnittstelle (Nutzungsschnittstelle) anzeigt und

- $ret_p^{R_S} \in DNODE^{R_S}$, ein Dummy-Programmpunkt, der die Rückkehr einer Prozedur p der benötigten Schnittstelle (Nutzungsschnittstelle) anzeigt.

Für alle Prozeduren oder Funktionen $u \in I_S$ (in der Angebotsschnittstelle) eines Service S existiert ein:

- $init_u^{I_S} \in DNODE^{I_S}$ ein Dummy-Programmpunkt als neuer Dummy-Eintrittsprogrammpunkt der Prozedur oder Funktion u und

- $ret_u^{I_S} \in DNODE^{I_S}$ ein Dummy-Programmpunkt als neuer Dummy-Austrittsprogramm punkt der Prozedur oder Funktion u.

Die Menge der $DNODE_S$ eines Service S ergibt sich aus:

$$DNODE_S = DNODE^{R_S} \cup DNODE^{I_S} \; mit \; DNODE^{I_S} \cap DNODE^{R_S} = \emptyset$$

Definition 4.18 (Zusätzliche Programmpunkte - $NNODE_S$)**.** Sei $NNODE_S$ die Menge der zusätzlichen Programmpunkte eines beliebigen Service S, wobei $NNODE_S$, $DNODE_S$ und $NODE_S$ paarweise disjunkt sind.

Für alle Prozeduren oder Funktionen $u \in I_S$ (in der Angebotsschnittstelle) eines Service S existiert ein:

- $s_u \in NNODE_S$ ein zusätzlicher, (künstlicher/unechter) Programmpunkt als neuer Eintrittsprogrammpunkt der Prozedur oder Funktion u und

- $r_u \in NNODE_S$ ein zusätzlicher Programmpunkt als neuer Austrittsprogrammpunkt der Prozedur oder Funktion u.

Definition 4.19 (Dummy-Speicher σ^{dy})**.** Sei $\sigma^{dy} : VAR \to (INT \cup BOOL)$ die Speicherfunktion, die den Dummy-Speicher kennzeichnet. Der leere Dummy-Speicher wird mit σ_0^{dy} gekennzeichnet. Der Dummy-Speicher für Aufrufe von Prozeduren oder Funktionen über Servicegrenzen hinaus wird mit σ_i^{dy} gekennzeichnet, wobei

- $\sigma_i^{dy}(p) = \Phi$, und Φ speichert die Anzahl der Aufrufe asynchroner Prozeduren oder Funktionen und p ist die asynchron aufgerufene Prozedur oder Funktion,

- $\sigma_i^{dy}(x_l) = z_l$ mit $l = \{0, ..., n\}$ und $n \in \mathbb{N}$ die Parameterübergabe ermöglicht,

mit $DB_i = \{p, x_0, \ldots, x_n\}, n \in \mathbb{N}$ und $WB_r = \mathbb{N} \cup \mathbb{B}$.

Die Menge der Aufruf-Dummy-Speicher ist $DUMMY_i^A$ für einen Service A.

Der Dummy-Speicher für die Rückkehr von Prozedur- und Funktionsaufrufen über Servicegrenzen hinaus, wird mit σ_r^{dy} gekennzeichnet, wobei

- $\sigma_r^{dy}(p) = n$ mit $n \in \mathbb{N}$ zur Speicherung der Anzahl der Aufrufe von asynchronen Prozeduren verwendet wird und p die asynchron aufgerufene Prozedur oder Funktion ist und

- $\sigma_r^{dy}(rval) = z$ mit $rval$ die Speicherung des Rückgabewerts bei Funktionsaufrufen ermöglicht.

mit $DB_r = \{p, rval\}$ und $WB_r = \mathbb{N} \cup \mathbb{B}$.

Die Menge der Rückkehr-Dummy-Speicher ist $DUMMY_r^A$ für einen Service A.

Definition 4.20 (Kaktuskeller-Elemente eines Services - $DSFRAME$)**.** Die Menge aller Kaktuskeller-Elemente $DSFRAME_S$ eines Service S ergibt sich aus den Platzhaltern $DFRAME_S$, den zusätzlichen Kaktuskeller-Elementen $NFRAME_S$ und den regulären Kaktuskeller-Elementen $SFRAME_S$ des Services S.

$DSFRAME = DFRAME_S \cup NFRAME_S \cup SFRAME_S$,
wobei $DFRAME_S$, $SFRAME_S$ und $NFRAME_S$ paarweise disjunkt sind.

Beispiel 4.3 (Platzhalter am Beispiel)**.** Anhand von Service A in Abbildung 4.14 sollen die für die Komposition benötigten Dummy-Programmpunkte erläutert werden. Der Service A stellt für beliebige Services in einem Service-orientierten System die Implementierung der synchronen Prozedur a über die Angebotsschnittstelle I_A zur Verfügung. Für den Eintrittspunkt i_a wird in der operationalen Semantik ein neuer Dummy-Programmpunkt $init_a^{I_A}$ benötigt, sowie ein neuer zusätzlicher Programmpunkt $s_a^{I_A}$ und für die *return*-Anweisung am Programmpunkt re_a wird ebenfalls ein neuer zusätzlicher Programmpunkt $r_a^{I_A}$ als Austrittspunkt, und der Dummy-Programmpunkt $ret_a^{I_A}$ eingeführt. Beide Dummy-Programmpunkte sind Angebots-Dummy-Programmpunkte aus der Menge $DNODE_a^{I_A}$ des Service A für die Prozedur a.

Der Service A benötigt eine Implementierung der asynchronen Funktion b, vgl. mit der benötigten Schnittstelle R_A. Im Programmpunkt q_{a1} ruft Service A die asynchrone Prozedur b auf, die er nicht selbst implementiert, siehe R_A. Für den aus q_{a1} aufgerufenen Programmpunkt wird der Dummy-Programmpunkt $init_b^{R_A}$ eingeführt. Dabei steht der Dummy-Programmpunkt $init_b^{R_A}$ für den Eintrittsprogrammpunkt einer Prozedur b in einem beliebigen Service S, der b implementiert und b über eine angebotene Schnittstelle I_S zur Verfügung stellt. Der Dummy-Programmpunkt $ret_b^{R_A}$ wird benötigt, um die Synchronisationsanweisung im Programmpunkt q_{a3} nach der Komposition simulieren zu können. Der nach unserem Ausführungsmodell benötigte Platzhalter für die Synchronisation der asynchron aufgerufenen Prozedur b mit der *return*-Anweisung der aufrufenden Prozedur a im Programmpunkt re_a von Service A wird auch durch den

Platzhalter $ret_b^{R_A}$ realisiert. Diese Dummy-Programmpunkte sind benötigte Dummy-Programmpunkte (Nutzungsschnittstelle) aus der Menge $DNODE_b^{R_A}$ des Service A für die Prozedur b. Die Menge der Dummy-Programmpunkte $DNODE_A$ für den Service A ist $DNODE_a^{I_A} \cup DNODE_b^{R_A}$.

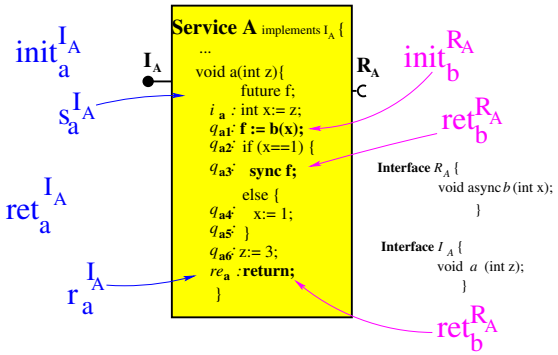


Abbildung 4.14.: Service A mit den Schnittstellen R_A, I_A und den zusätzlichen Programmpunkten $s_a^{I_A}$ und $r_a^{I_A}$ sowie den Dummy-Programmpunkten $init_b^{I_A}$ und $ret_b^{I_A}$ für die Angebotsschnittstelle und die Dummy-Programmpunkte $init_b^{R_A}$ und $ret_b^{R_A}$ für die Nutzungsschnittstelle.

Daraus ergeben sich für Service A folgende Programmpunkte, zusätzliche Programmpunkte und Dummy-Programmpunkte $DNODE_A$:

- $init_a^{I_A} \in DNODE^{I_A}$ als ein angebotener Dummy-Programmpunkt (Angebotsschnittstelle),
- $ret_a^{I_A} \in DNODE^{I_A}$ als ein angebotener Dummy-Programmpunkt (Angebotsschnittstelle),
- $init_b^{R_A} \in DNODE^{R_A}$ als ein benötigter Dummy-Programmpunkt (Nutzungsschnittstelle),
- $ret_b^{R_A} \in DNODE^{R_A}$ als ein benötigter Dummy-Programmpunkt (Nutzungsschnittstelle),
- $s_a^{I_A} \in NNODE_A$ als ein zusätzlicher Programmpunkt,
- $r_a^{I_A} \in NNODE_A$ als ein zusätzlicher Programmpunkt und
- $\{i_a, q_{a1}, q_{a2}, \ldots, q_{a6}, re_a\} = NODE$ als reguläre Programmpunkte.

Daraus ergeben sich für Service A folgende Platzhalter $DFRAME_A$:

- $(init_a^{I_A}, \sigma^{dy}) \in DFRAME^{I_A}$ als ein Platzhalter der Angebotsschnittstelle für den Eintrittspunkt,

- $(ret_a^{I_A}, \sigma_0^{dy}) \in DFRAME^{I_A}$ als ein Platzhalter der Angebotsschnittstelle für den Austrittspunkt, mit dem leeren Speicher, da a eine Prozedur ist,

- $(init_b^{R_A}, \sigma^{dy}|_{b,x}^{n,eval^{\sigma}(x)}) \in DFRAME^{R_A}$ als ein Platzhalter für die Nutzungsschnittstelle (mit dem Dummy-Speicher σ^{dy}, wobei b der Name der aufgerufenen Prozedur ist, $n \in \mathbb{N}$ ist die Anzahl asynchroner Aufrufe Φ) und in der Variable x wird der Wert von x bezüglich des Speichers σ geschrieben,

- $(ret_b^{R_A}, \sigma^{dy}|_b^n) \in DFRAME^{R_A}$ als ein Platzhalter für die Nutzungsschnittstelle ($n \in \mathbb{N}$ ist die Anzahl asynchroner Aufrufe (Φ)).

Darüber hinaus ergeben sich für Service A folgende zusätzliche Kaktuskeller-Elemente $NFRAME_A$:

- $(s_a^{I_A}, \sigma^{dy}) \in NFRAME_A$ als ein zusätzliches Kaktuskeller-Element (Angebotsschnittstelle),

- $(r_a^{I_A}, \sigma_0^{dy}) \in NFRAME_A$ als ein zusätzliches Kaktuskeller-Element mit dem zusätzlichen Austrittspunkt und dem leeren Speicher, da a eine Prozedur ist.

Mit Hilfe der Platzhalter und zusätzlichen Programmpunkte können nun die Regeln der operationalen Semantik aus Kapitel 4.2 so erweitert werden, dass sie das Verhalten von Prozedur- und Funktionsaufrufen über Servicegrenzen hinaus und damit das Verhalten von Services darstellen können. Die Regeln (PS_0), (PSR_0), (PS_1), (PSR_1), (PA_0), (PA_1), (PA_{sync}) und (PAR_{sync}) werden dementsprechend erweitert. Die aus der Anpassung resultierenden Regeln $(PS_0^{\rightarrow +})$, $(PSR_0^{\rightarrow +})$, $(PS_1^{\rightarrow +})$, $(PSR_1^{\rightarrow +})$, $(PA_0^{\rightarrow +})$, $(PA_1^{\rightarrow +})$, $(PA_{sync}^{\rightarrow +})$ und $(PAR_{sync}^{\rightarrow +})$ ermöglichen die Darstellung des Verhaltens von Services und service-orientierten Systemen.

Definition 4.21 (Regeln der Operationalen Semantik für Services)**.** Die Regeln der operationalen Semantik für Services ergeben sich aus den Regeln der Auswertungsrelation $\rightarrow$ der operationale Semantik der Beispielsprache $\mathcal{XYZ}$, definiert in Definition 4.8, und den Regeln der Auswertungsrelation $\rightarrow$ aus den Tabellen 4.15 bis 4.20.

Die Regeln der Auswertungsrelation $\rightarrow$ werden in den Abbildungen 4.15 bis 4.20 aufgelistet intuitiv erläutert.

Prozedur und Funktionsaufrufe ohne Parameter über Servicegrenzen hinaus, werden in Abbildung 4.15 betrachtet. Die Eintrittsprogrammpunkte der aufgerufenen Prozedur p (mit $p \in R_A$) wird durch $init_p^{R_A}$ ersetzt. Der Dummy-Speicher σ_i^{dy} ist dabei der

leere Speicher, der beim Aufruf übergeben wird. Der Speicher ist leer, da hier weder Parameter übergeben werden, noch future-Informationen gespeichert werden müssen. Der Rückkehr-Programmpunkt q' von p, wird um den Dummy-Programmpunkt $ret_p^{R_A}$ ersetzt. Die neuen Programmpunkte werden Dummy-Programmpunkte genannt und ermöglichen die Komposition des Verhaltens der Services, die miteinander verbunden werden. In Abbildung 4.15 spezifiziert der Dummy-Programmpunkt $init_p^{I_A}$ in der Regel $(PSR_0^{\rightarrow +})$ den Eintrittsprogrammpunkt der Prozedur p. Dieser Dummy-Programmpunkt kann über den im Grundlagenkapitel eingeführten Kompositionsmechanismus, an einen Dummy-Programmpunkt $init_p^{R_X}$ einer Prozedur oder Funktion aus einer angebotenen Schnittstellenbeschreibung R_X gebunden werden. Der Dummy-Austrittsprogrammpunkt $ret_p^{I_A}$ der Nutzungsschnittstelle wird analog mit dem Dummy-Austrittsprogrammpunkt der Angebotsschnittstelle gebunden werden.

Die Regeln $(PS_1^{\rightarrow +})$ und $(PSR_1^{\rightarrow +})$ kommen zur Anwendung, wenn das Verhalten von Funktionen mit Parameterübergabe über Servicegrenzen hinaus dargestellt werden soll. Die Regel $(PSR_1^{\rightarrow +})$ modelliert dabei das Verhalten, wenn die aufgerufene Funktion einen Rückgabewert besitzt. Die Aufrufe der Prozeduren in der Schnittstellenbeschreibung R_A führt zur Einführung der Dummy-Programmpunkte $init_p^{R_A}$ und $ret_p^{R_A}$. Der Dummy-Programmpunkt $init_p^{R_A}$ präsentiert dabei den Eintrittspunkt der aufgerufenen Funktion und $ret_p^{R_A}$ den Austrittspunkt der aufgerufenen Funktion p (analog zu $(PS_0^{\rightarrow +})$ und $(PSR_0^{\rightarrow +})$).

Die Regeln $(PA_0^{\rightarrow +})$ und $(PA_1^{\rightarrow +})$ beschreiben das Verhalten von über Servicegrenzen hinaus aufgerufene asynchrone Prozeduren. Im Unterschied zu synchron aufgerufenen Prozeduren, müssen für asynchron aufgerufene Prozeduren Informationen bezüglich des futures gespeichert werden. In der Umgebung σ wird dazu unter der future-Programmvariable das Tupel (p, n) gespeichert, wobei p der Name der aufgerufenen Prozedur entspricht und n die Anzahl Aufrufe asynchroner Prozeduren und Funktionen ist. Bevor eine asynchrone Prozedur vom Aufrufer beim Erreichen der *return*-Anweisung synchronisiert werden kann, muss die Bedingung erfüllt sein, dass der Aufrufer auch der tatsächliche Aufrufer von p ist, vgl. $\sigma(f) = (p, \sigma^{dy}(p))$.

Zur Darstellung der Synchronisation von über Servicegrenzen hinaus aufgerufene Prozeduren und Funktionen, werden die Regeln $(PA_{sync}^{\rightarrow +})$ und $(PAR_{sync}^{\rightarrow +})$ benötigt. Wenn im Speicher des Aufrufers der zu synchronisierende Prozess, der mit dem $future f$ mit n aufgerufen wurde, dann kann synchronisiert werden. In der Regel $(PAR_{sync}^{\rightarrow +})$ wird zusätzlich noch der Rückgabewert von p in der Variable *rval* nach Def. 4.19 gespeichert und anschließend in dem Speicher σ' unter der Programmvariable x gespeichert.

Nachdem die operationale Semantik der Beispielsprache erweitert wurde, um auch das Verhalten von Prozedur- und Funktionsaufrufen über Servicegrenzen hinaus zu modellieren, wird im Folgenden die Relation der Zustandsübergänge für Prozeduren eines Service S, die in der angebotenen Schnittstelle zur Verfügung stehen, definiert. Es wer-

$(PS_0^{\rightarrow+})$	$\dfrac{\overline{(q,\sigma) \rightarrow (init_p^{R_A}, \sigma_i^{dy}).(q',\sigma)}}{(ret_p^{R_A}, \sigma_r^{dy}).(q',\sigma) \rightarrow (q',\sigma)}$,	
falls	**Interface** $R_A\{void\ p();\}$	$falls$ $q:$ $p()$ $q':$ $s_1,$ $\ldots$
$(PSR_0^{\rightarrow+})$	$\dfrac{\overline{(q,\sigma) \rightarrow (init_p^{R_A}, \sigma_i^{dy}).(q',\sigma)}}{(ret_p^{R_A}, \sigma_r^{dy}).(q',\sigma) \rightarrow (q', \sigma\vert_y^{eval^{\sigma_r^{dy}}(rval)})}$,	
falls	**Interface** $R_A\{int\ p();\}$	$falls$ $q:$ $y := p()$ $q':$ $s_1,$ $\ldots$

Wobei q, $q' \in NODE$ und $init_p^{R_A}$, $ret_p^{R_A} \in DNODE^{R_A}$ sind und $\sigma_i^{dy}, \sigma_r^{dy}$ Dummy-Speicher sind. σ ist ein regulärer Speicher.

Abbildung 4.15.: Regeln der Operationalen Kaktuskeller-Semantik für synchrone Prozedur- und Funktionsaufrufe ohne Parameterübergabe über Servicegrenzen hinaus.

$(PS_1^{\rightarrow+})$	$\dfrac{\overline{(q,\sigma) \rightarrow (init_p^{R_A}, \sigma_i^{dy}\vert_{x_0,\ldots,x_n}^{eval^\sigma(y_0),\ldots,eval^\sigma(y_n)}).(q',\sigma)}}{\overline{(ret_p^{R_A}, \sigma_r^{dy}).(q',\sigma) \rightarrow (q',\sigma)}}$,	
falls	**Interface** $R_A\{void\ p(int\, x_0, \ldots, int\, x_n);\}$	$falls$ $q:$ $p(y_0, \ldots, y_n)$ $q':$ s_1 $\ldots$
$(PSR_1^{\rightarrow+})$	$\dfrac{\overline{(q,\sigma) \rightarrow (init_p^{R_A}, \sigma_i^{dy}\vert_{x_0,\ldots,x_n}^{eval^\sigma(y_0),\ldots,eval^\sigma(y_n)}).(q',\sigma)}}{(ret_p^{R_A}, \sigma_r^{dy}).(q',\sigma) \rightarrow (q', \sigma\vert_y^{eval^{\sigma_r^{dy}}(rval)})}$,	
falls	**Interface** $R_A\{int\ p(int\, x_0, \ldots, int\, x_n);\}$	$falls$ $q:$ $y := p(y_0, \ldots, y_n)$ $q':$ s_1 $\ldots$

Wobei q, $q' \in NODE$, $init_p^{R_A}, ret_p^{R_A} \in DNODE^{R_A}$ und σ ist ein Speicher und σ_i^{dy}, σ_r^{dy} Dummy-Speicher sind.

Abbildung 4.16.: Regeln der Operationalen Kaktuskeller-Semantik für Funktionsaufrufe mit Parameterübergabe über Servicegrenzen hinaus.

$(PA_0^{\rightarrow+})$	$\dfrac{\overline{(q,\sigma) \to (q',\sigma'|_f^{(p,n)})||(init_p^{R_A}, \sigma_i^{dy}|_p^n)}, \quad \sigma(f) = (p, \sigma_r^{dy}(p))}{(q'',\sigma) \parallel (ret_p^{R_A}, \sigma_r^{dy}) \to (q'',\sigma)}$	
falls	**Interface** $R_A\{void\,\mathbf{async}\,p();\}$	$\begin{array}{rl} & falls \\ & futuref; \\ q: & f := p(); \\ q': & s_1, \ldots \\ q'': & return;\} \\ & \ldots \end{array}$
$(PA_1^{\rightarrow+})$	$\dfrac{\overline{(q,\sigma) \to (q',\sigma'|_f^{(p,n)})||(init_p^{R_A}, \sigma_i^{dy}|_{p,x_0,\ldots,x_n}^{n,eval^{\sigma_i^{dy}}(v_0),\ldots,eval^{\sigma_i^{dy}}(v_n)})}, \quad \sigma(f) = (p, \sigma_r^{dy}(p))}{(q'',\sigma) \parallel (ret_p^{R_A}, \sigma_r^{dy}) \to (q'',\sigma)}$	
falls	**Interface** $R_A\{void\,\mathbf{async}\,p(int\,x_0, \ldots, int\,x_n);\}$	$\begin{array}{rl} & falls \\ & futuref; \\ q: & f := p(v_0, \ldots, v_n) \\ q': & s_1, \ldots \\ q'': & return;\} \\ & \ldots \end{array}$

Wobei q, q', $q'' \in NODE$ und Speicher sind σ, σ' und σ_i^{dy}, σ_r^{dy} Dummy-Speicher sind und $n \in \mathbb{N}$.

Abbildung 4.17.: Regeln der Operationalen Kaktuskeller-Semantik für asynchrone Prozeduraufrufe mit und ohne Parameterübergabe über Servicegrenzen hinaus.

$(PA^{\rightarrow +}_{sync})$	$\dfrac{\sigma(f) = (p, \sigma_r^{dy}(p))}{(q'', \sigma) \parallel (ret_p^{R_A}, \sigma_r^{dy}) \rightarrow (q''', \sigma)}$	
falls	**Interface** $R_A\{void\,\mathbf{async}\,p(int\,v_0, \ldots, int\,v_n);\}$	$\ldots$ $q^f:$ $futuref;$ $\ldots$ $q:$ $f := p(x_0, \ldots, x_n)$ $q':$ $s_1, \ldots$ $q'':$ $sync\,f;$ $q''':$ $s_2, \ldots$ $q'''':$ $return\,z;\}$ $\ldots$
$(PAR^{\rightarrow +}_{sync})$	$\dfrac{\sigma(f) = (p, \sigma_r^{dy}(p))}{(q'', \sigma)\|\|(ret_p^{R_A}, \sigma_r^{dy}) \rightarrow (q''', \sigma'|_x^{eval^{\sigma^{dy}}(rval)})}$	
falls	**Interface** $R_A\{int\,\mathbf{async}\,p(int\,v_0, \ldots, int\,v_n);\}$	$\ldots$ $q:$ $p(x_0, \ldots, x_n)$ $\ldots$ $q^f:$ $futuref;$ $q':$ $s_1, \ldots$ $q'':$ $x := sync\,f;$ $q''':$ $s_2, \ldots$ $q'''':$ $return;$ $\ldots$

Wobei q, q', q'', q''', q'''', q^f, $\in NODE$, $ret_p^{R_A} \in DNODE^{R_A}$, σ, σ' Speicher sind und σ_r^{dy} ein Dummy-Speicher ist.

Abbildung 4.18.: Regeln der Operationalen Kaktuskeller-Semantik zur Synchronisation von über Servicegrenzen hinaus aufgerufene asynchrone Funktionen.

den pro Signatur in einer angebotenen Schnittstelle zwei zusätzliche Programmpunkte eingeführt:

- $s_u^{I_S}$, vgl. Definition 4.18 und
- $r_u^{I_S}$, vgl. Definition 4.18.

Zu den überarbeiteten Regeln kommen die Regeln aus den Abbildungen 4.19 bis 4.20. Sie verbinden die zusätzlichen Programmpunkte in den angebotenen Schnittstellen mit den Eintritts-Programmpunkt bzw. Austritts-Programmpunkt einer Prozedur oder Funktion.

In der Abbildung 4.19 werden die Regeln eingeführt, die benötigt werden, um die zusätzlichen Kaktuskeller-Elemente $s_p^{I_A}$ mit dem tatsächlichen Kaktuskeller-Element, welches den Eintrittsprogrammpunkt, dem ersten Programmpunkt im Rumpf der Prozedur p und den entsprechenden Speicher repräsentiert, zu verknüpfen. Die zusätzlichen Kaktuskeller-Elemente $(s_p^{I_A}, \sigma_i^{dy})$ werden durch die Platzhalterfunktion $\mathcal{D}$ mit dem Eintritts-Dummy-Programmpunkt $(init_p^{I_A})$ verbunden.

In Abbildung 4.20 werden die Regeln für die Übergänge vom Kaktuskeller-Element des letzten Programmpunktes re der Prozedur oder Funktion p einer Angebotsschnittstelle I_A eines Service A zum zusätzlichem Kaktuskeller-Element $(r_p^{I_A}, \sigma^{dy})$ beschrieben. Die Platzhalterfunktion $\mathcal{D}$ verknüpft bei einer Komposition den Rückkehr-Dummy-Platzhalter mit dem zusätzlichen Kaktuskeller-Element des Service der Angebotsschnittstelle. Bei den Rückkehr-Dummy-Speicher σ_r^{dy} handelt es sich um einen leeren Speicher, wenn die Prozedur p keinen Rückgabewert hat und synchron aufgerufen wird. Der Rückkehr-Dummy-Speicher beinhaltet den Wert der Variable $rval$, wenn p eine Funktion ist und damit einen Wert zurück gibt (vgl. $(SPP_r^{\rightarrow +})$).

Definition 4.22 (Semantik einer Prozedur in Angebotsschnittstellen)**.** Sei I_S eine Angebotsschnittstelle eines Service S mit der Prozedur oder Funktion $p \in I_S$ mit einem eindeutigen Eintritts-Programmpunkt $i_p \in DNODE_S^{I_A}$ und Austritts-Programmpunkt $re_p \in DNODE_S^{I_A}$. Ein Platzhalter oder Dummy für den Aufruf von p ist das Tupel $(init_p^{I_A}, \sigma_i^{dy}) \in DFRAME_S^{I_A}$ und für die Rückkehr von p das Tupel $(ret_p^{I_A}, \sigma_r^{dy}) \in DFRAME_S^{I_A}$, wobei $\sigma_i^{dy} \in DUMMY_i^S$ als Dummy-Speicher für future-Informationen und die Übergabe von Parametern genutzt wird und der Dummy-Speicher $\sigma_r^{dy} \in DUMMY_r^S$ den Rückgabewert und bei asynchronen Aufrufen future-Informationen enthält. Neben den Platzhaltern werden zusätzliche Programmpunkte $s_p \in NNODE_S$ und $r_p \in NNODE_S$ für p eingeführt, die die zusätzlichen Kaktuskeller-Elemente $(s_a^{I_A}, \sigma_i^{dy}) \in NFRAME_S$ und $(r_a^{I_A}, \sigma_r^{dy}) \in NFRAME_S$ bilden.

Sei R_S eine Angebotsschnittstelle eines Service S mit der Prozedur oder Funktion $u \in R_S$. Ein Platzhalter oder Dummy für den Aufruf von p ist das Tupel $(init_p^{R_S}, \sigma_i^{dy}) \in$

$(SP_i^{\rightarrow +})$	**Interface** $I_A\{void\|int\ p();\}$
	$\overline{(s_p^{I_A}, \sigma_0^{dy}) \rightarrow (i_p, \sigma_0)}$
$(SPP_i^{\rightarrow +})$	**Interface** $I_A\{void\|int\ p(int\, z_0, \ldots, int\, z_n);\}$
	$\overline{(s_p^{I_A}, \sigma_i^{dy}) \rightarrow (i_p, \sigma\|_{z_0,\ldots,z_n}^{eval^{\sigma_i^{dy}}(x_0),\ldots,eval^{\sigma_i^{dy}}(x_n)})}$
$(AP_i^{\rightarrow +})$	**Interface** $I_A\{void\|int$ **async** $p();\}$
	$\overline{(s_p^{I_A}, \sigma_i^{dy}) \rightarrow (i_p, \sigma\|_{p}^{eval^{\sigma_i^{dy}}(p)})}$
$(APP_i^{\rightarrow +})$	**Interface** $I_A\{void\|int$ **async** $p(int\, z_0, \ldots, int\, z_n);\}$
	$\overline{(s_p^{I_A}, \sigma_i^{dy}) \rightarrow (i_p, \sigma\|_{p,z_0,\ldots,z_n}^{eval^{\sigma_i^{dy}}(p),eval^{\sigma_i^{dy}}(x_0),\ldots,eval^{\sigma_i^{dy}}(x_n)})}$

Wobei $s_p^{I_A}$ Dummy-Eintritts-Programmpunkte, i_p der erste Programmpunkt im Rumpf der Prozedur oder Funktion p ist, σ_i^{dy} ein Dummy-Speicher, σ ein regulärer Speicher sei. I_A sei die angebotene Schnittstelle von Service A, der p implementiert.

Abbildung 4.19.: Regeln der Operationalen Kaktuskeller-Semantik für Services für den Eintritt.

$DFRAME_S^{R_S}$ und für die Rückkehr von p das Tupel $(ret_p^{R_S}, \sigma_r^{dy}) \in DFRAME_S^{R_S}$, wobei $\sigma_i^{dy} \in DUMMY_i^S$ als Dummy-Speicher für future-Informationen und die Übergabe von Parametern genutzt wird und der Dummy-Speicher $\sigma_r^{dy} \in DUMMY_r^S$ den Rückgabewert und bei asynchronen Aufrufen future-Informationen enthält.

Die Semantik von S bezüglich einer Prozedur p einer Angebotsschnittstelle I_S ist das Zustandsübergangssystem $[\![S^{I_{S_p}}]\!] = (Z_{S^{I_{S_p}}}, Q_0, \rightarrow_S, F, \mathcal{D})$, wobei:

- $Z_{S^{I_{S_p}}} \subseteq PEX(DSFRAME_S)$ eine Menge von Kaktuskellern,
- $Q_0 = \{(s_p^{I_S}, \sigma_i^{dy})\}$ der Anfangszustand (zusätzliche Kaktuskeller-Element $NFRAME_S$) mit leeren Speicher, wobei $s_p^{I_S} \in NNODE_i^{I_S}$ und $\sigma_i^{dy} \in DUMMY_i^{I_S}$ ein Aufruf-Dummy-Speicher ist,
- $\rightarrow_{S_p} \subseteq Z \times Z$ ist die Zustandsübergangsrelation auf Grundlage der Ableitungsrelationen der operationalen Semantik (Abb. 4.1 bis 4.20),
- $F = \{(r_p^{I_S}, \sigma_r^{dy})\}$ der Endzustand (zusätzliche Kaktuskeller-Element $NFRAME_S$) als Austrittspunkt, wobei $r_p^{I_S} \in NNODE_r^{I_S}$ und $\sigma_r^{dy} \in DUMMY_r^{I_S}$ (Rückkehr-Dummy-Speicher) und,
- $\mathcal{D}_{S_p} : ((DUMMY_S^{I_S}, \sigma_x^{dy})) \rightarrow (NNODE_S^{I_S}, \sigma_x^{dy})$ mit $x \in \{i, r\}$ die Platzhalterfunktion.

Definition 4.23 (Semantik Prozedur main)**.** Die Semantik der *main*-Prozedur m eines Klientenservice K ist definiert durch das Zustandsübergangssystem $[\![K_m]\!] = (Z_{K_m}, Q_0, \rightarrow_S, F, \mathcal{D})$, wobei:

- $Z_{K_m} \subseteq PEX(DSFRAME_K)$ eine Menge von Kaktuskellern,
- $Q_0 = \{(q_m, \sigma_0)\}$ der Anfangszustand mit leeren Speicher,
- $\rightarrow_{K_m} \subseteq Z \times Z$ die Zustandsübergangsrelation auf Grundlage der Ableitungsrelationen der operationalen Semantik (Abb. 4.1 bis 4.20),
- $F = \{(q_f, \sigma)\}$ der Endzustand (q_f ist Programmpunkt der *return*-Anweisung im Rumpf von *main*) und
- $\mathcal{D}$ nicht definiert ist.

Bemerkung 4.7. *Besitzt ein Klientenservice Angebotsschnittstellen, so wird die Semantik des Klientenservices bezüglich den in der Angebotsschnittstellen vorhandenen Prozeduren oder Funktionen nach Def. 4.22 angegeben beziehungsweise erweitert.*

Definition 4.24 ($Platzhalterfunktion - \mathcal{D}$)**.** Sei $\mathcal{D}$ die Platzhalterfunktion eine Abbildungsfunktion mit $\mathcal{D} : DFRAME \rightarrow NFRAME$.

Die Platzhalter vom Service A einer Angebotsschnittstelle I_A bilden auf die zusätzlichen Kaktuskeller-Elemente der Angebotsschnittstelle I_A des Service S ab. Dabei steht das i und das r im Index jeweils für den Eintrittspunkt bzw. Austrittspunkt der aufgerufenen Prozedur oder Funktion p der Angebotsschnittstelle.

Definition 4.25 (Semantik von Services)**.** Sei S ein Service mit Angebotsschnittstellen I_{S_1}, ..., I_{S_n}. Die Semantik von S ist die Menge aller Zustandsübergangssysteme ihrer Angebotsschnittstellen $[\![S]\!] = [\![S^{I_{S_1}}]\!] \cup \ldots \cup [\![S^{I_{S_n}}]\!]$ mit $n \in \mathbb{N}$.

Die Menge der Zustandsübergangssysteme $[\![S^{I_S}]\!]$ einer Angebotsschnittstelle I_S ergibt sich aus der Menge der Zustandsübergangssysteme aller Prozeduren oder Funktionen p_1, ..., p_n der Angebotsschnittstellen I_S: $[\![I_S]\!] = [\![I_{S_{p_1}}]\!] \cup \ldots \cup [\![I_{S_{pn}}]\!]$ mit $n \in \mathbb{N}$.

$[\![I_S]\!] = (Z_{I_S}, Q_0, \rightarrow_S, F, \mathcal{D})$, wobei:

- $Z_{I_S} \subseteq PEX(DSFRAME_S)$ eine Menge von Kaktuskellern,
- $Q_0 = \{(s_p^{I_S}, \sigma_i^{dy}) : p \in I_S\}$ die Anfangszustände aller Prozeduren p in I_S (zusätzliche Kaktuskeller-Element $NFRAME_S$),
- $\rightarrow_{S^{I_S}} = \bigcup_{i=1}^{n} \rightarrow_{I_{S_{p_i}}}$ ist die Menge der Regeln der Zustandsübergangsrelationen aller Prozeduren und Funktion p_i in der Angebotsschnittstelle I_S,
- $F = \{(r_p^{I_S}, \sigma_r^{dy}) : p \in I_S\}$ der Endzustände aller Prozeduren p in I_S,
- $\mathcal{D}_\mathcal{S} : ((DUMMY_S^{I_S}, \sigma_x^{dy})) \rightarrow (NNODE_S^{I_S}, \sigma_x^{dy})$ mit $x \in \{i, r\}$ die Platzhalterfunktion.

Nach der Definition 4.25 muss das Verhalten eines jeden Services durch die Aufstellung der Relationen der Zustandsübergänge mit Hilfe der vorgestellten operationalen Kaktuskeller-Semantik aufgestellt werden. Über Servicegrenzen hinausgehende Prozedur- oder Funktionsaufrufe werden mit Platzhaltern versehen, ebenso die Ein- und Austrittspunkte von implementierten Prozeduren und Funktionen, die nach außen zur Verfügung gestellt werden.

4.3.2. Komposition von Services

Die Komposition von Services zu einem Service-orientierten System wird im Grundlagenkapitel 3.1.2 beschrieben. Die Services werden über ihre Angebots- und Nutzungsschnittstellen miteinander verbunden. Im Folgenden wird die Komposition der operationalen Semantik von Services beschrieben.

Komposition der operationalen Semantik von Services

Das Verhalten eines Service-orientierten Systems kann mit Hilfe der Regeln der operationalen Semantik dargestellt werden. Dafür müssen die Regeln der Ableitungsrelationen der einzelnen Services bezüglich der aufgerufenen Prozeduren und Funktionen in den Angebots- und Nutzungsschnittstellen im Service-orientierten System miteinander verknüpft werden. Das Zustandsübergangssystem eines Service-orientierten Systems ergibt sich aus der Komposition der Zustandsübergangsrelationen der Services S_i in die Relation der Zustandsübergänge des Service-orientierten Systems SoS und der Ausführung der Platzhalterfunktionen zu r Auflösung der entsprechenden Platzhalter.

Das Zustandsübergangssystem eines Service-orientierten Systems $\rightarrow_{SoS}$ ist folgendermaßen definiert:

Definition 4.26 (Komposition der Op. Semantik von Services)**.** Sei die Komposition der Regelsysteme der operationalen Semantik der Services $S_1, \ldots, S_n$ und Klientenservices $K_1, \ldots, K_l$ das Regelsystem der operationalen Semantik des Service-orientierten Systems SoS ein Quintupel $[\![SoS]\!] = (Z_{SoS}, Q_{SoS}, F_{SoS}, \rightarrow_{SoS}, \mathcal{D}_{SoS})$ mit:

- $Z_{SoS} = \bigcup_{i=1}^{n} DSFRAME_{S_i}$, $n \in \mathbb{N}$ wobei S_i alle Service von SoS sind,
- $Q_{SoS} = \{(q_m, \sigma_0) : m \in K_1, \ldots, K_l\}$, $Q_0 \subseteq DSFRAME_{K_i}$ als Anfangszustand eines Klienten K_i im Service-orientierten System,
- $F_{SoS} = \bigcup_{i=1}^{l} F_{K_i}$,
- $\mathcal{D}_{SoS} = \bigcup_{i=1}^{n} \mathcal{D}_{S_i}$ wobei $\mathcal{D}_{S_i}(DFRAME) : NODE_{S_{p_i}}^{I_S}$ die Menge aller Platzhalterfunktionen der beteiligten Services S_i ist und
- $\rightarrow_{SoS} = \bigcup_{i=1}^{n} \rightarrow_{S_i}$, $n \in \mathbb{N}$ die Menge über alle Zustandsübergangsrelationen auf Basis der operationalen Semantik.

Das Aufstellen der operationalen Semantik für einzelne Services, die Komposition der Verhaltensregeln und die Darstellung des Verhaltens eines Service-orientierten Systems soll im weiteren Verlauf anhand eines Beispiels verdeutlicht werden.

4.4. Verhalten eines Service-orientierten Systems am Beispiel

Eine Reihe von Services, die zu einem Service-orientierten System verknüpft werden können, sind in Abbildung 4.21 zu sehen.

Beispiel 4.4. Der Service M ist der Klientenservice, da er eine *main*-Methode enthält. Service M benötigt die Schnittstelle R_M. Service B implementiert die synchrone Prozedur a, die über die angebotene Schnittstelle I_A zur Verfügung gestellt wird. Service B ruft in a die asynchronen Funktionen b und c auf, die nicht implementiert im Service B vorliegen und sich daher als Signatur in der Schnittstelle R_A befinden. Die Services B und C stellen die Implementierungen der asynchronen Prozeduren b und c zur Verfügung, vgl. Schnittstellen I_B und I_C.

Im Folgenden wird über die definierten Regeln der operationalen Kaktuskeller-Semantik für jeden Service das Regelsystem zur Darstellung des Verhaltens aufgestellt.

4.4.1. Verhalten der Services

Auf Basis der vorgestellten Regeln in Abbildung 4.1 bis 4.20 wird das Verhalten für jedem Service in Abbildung 4.21 als Zustandsübergangssystem erstellt.

Verhalten von Service M

Service M verfügt über eine Nutzungsschnittstelle R_M und ist ein Klientenservice. In Abbildung 4.22 ist der Quellcode von Service M zu sehen. Die dazugehörigen Regeln $\rightarrow_M$ sind in der zweiten Spalte aufgeführt. Diese Regeln zusammen mit den Zuständen, inklusive Anfangszustand und Endzustand, sowie den Platzhaltern ergeben das Zustandsübergangssystem $Serv_M$ für den Service M.

Das Zustandsübergangssystem $[\![Serv_M]\!] = (Z_M, \rightarrow_M, Q_0, F, \mathcal{D})$, für den Service M ist gegeben durch:

- $Z_M = PEX(DSFRAME_M)$,

- $\rightarrow_M$ die Menge von Zustandsübergangsregeln in Abbildung 4.22,
- $Q_0 = \{(q_m, \sigma_0)\}$ ist der Anfangszustand mit leeren Speicher,
- $F = \{(q_f, \sigma)\}$ und
- $\mathcal{D}_M$ nicht definiert.

Da der Service M über keine angebotenen Schnittstellen verfügt, wird keine Platzhalterfunktion benötigt und somit ist $\mathcal{D}$ nicht definiert.

Verhalten von Service A

Service A bietet die Schnittstelle I_A mit der Implementierung der Prozedur a an. Neben der angebotenen Schnittstelle I_A benötigt der Service A die Schnittstelle R_A. Die Zustandsübergangsregeln sind in Abbildung 4.23 zu finden.

Das Verhalten von Service A wird durch das Zustandsübergangssystem $[\![Serv_A]\!] = (Z_A, \rightarrow_A, Q_0, F, \mathcal{D}_\mathcal{A})$ beschrieben:

- $SFRAME_A = PEX(DSFRAME_A)$,
- $\rightarrow_A$ die Menge von Zustandsübergangsregeln in Abbildung 4.23,
- $Q_0 = \{(s_a^{I_A}, \sigma_i^{dy})\}$ ist der Anfangszustand mit dem Dummy-Speicher, wobei $(s_a^{I_A}, \sigma_i^{dy}) \in DFRAME_A^{I_A}$
- $F = (r_a^{I_A}, \sigma_r'^{dy})$ ist der Endzustand mit dem Dummy-Speicher σ_r^{dy}, wobei $(r_a^{I_A}, \sigma_r^{dy}) \in DFRAME_A^{I_A}$.

Platzhalterfunktion für Service A:

$$\mathcal{D}_A(init_a^{I_A}, \sigma_i^{dy}) = (s_a, \sigma_i^{dy}) \tag{4.1}$$

$$\mathcal{D}_A(ret_a^{I_A}, \sigma_r^{dy}) = (r_a, \sigma_r^{dy}) \tag{4.2}$$

Der Service A bietet eine Implementierung von a über die Angebotsschnittstelle I_A an. Um die Komposition zu ermöglichen werden die zusätzlichen Eintrittspunkte $(s_a^{I_A}, \sigma_i^{dy})$ als neuer Eintrittspunkt (Anfangszustand) und $(r_a^{I_A}, \sigma_r^{dy})$ als zusätzlicher Austrittspunkt und damit Endzustand von Service A eingeführt.

Verhalten von Service B

Der Quellcode von Service B wird in Abbildung 4.24 gezeigt.

Die Semantik von Service B beschreibt das Zustandsübergangssystem $[\![Serv_B]\!] = (Z_B, \rightarrow_B, Q_0, F, \mathcal{D}_\mathcal{B})$:

- $SFRAME_B = PEX(DSFRAME_B)$,
- $\rightarrow_B$ die Menge von Zustandsübergangsregeln in Abbildung 4.24,
- $Q_0 = \{(s_{B_b}, \sigma_i^{dy})\}$ ist der Anfangszustand (zusätzlicher Eintrittspunkt für b),
- $F_{B_b} = \{(r_{B_b}, \sigma_r^{dy})\}$ ist Endzustand (zusätzlicher Austrittspunkt für b),

Die Platzhalterfunktion für Service B ist definiert durch:

$$\mathcal{D}_B(init_b^{I_B}, \sigma_i^{dy}) = (s_b, \sigma_i^{dy}) \tag{4.3}$$

$$\mathcal{D}_B(ret_b^{I_B}, \sigma_r^{dy}) = (r_b, \sigma_r^{dy}) \tag{4.4}$$

Verhalten von Service C

Der Quellcode der Prozedur c, die über die Angebotsschnittstelle I_C zur Verfügung gestellt wird, ist in Abbildung 4.25 zu sehen.

Die Semantik von Service C ist definiert durch das Zustandsübergangssystem $[\![Serv_C]\!] = (Z_C, \rightarrow_C, Q_0, F, \mathcal{D}_\mathcal{C})$:

- $Z_C = PEX(DSFRAME_C)$,
- $\rightarrow_C$ die Menge von Zustandsübergangsregeln in Abbildung 4.25,
- $Q_0 = \{(s_c, \sigma_c^{dy})\}$ ist der Anfangszustand (zusätzlicher Eintrittspunkt für c),
- $F = \{(r_c, \sigma_c'^{dy})\}$ ist der Endzustand (zusätzlicher Austrittspunkt für c),

Die Platzhalterfunktion für Service C ist definiert durch:

$$\mathcal{D}_C(init_c^{I_C}, \sigma_i^{dy}) = (s_c, \sigma_i^{dy}) \tag{4.5}$$

$$\mathcal{D}_C(ret_c^{I_C}, \sigma_r^{dy}) = (r_c, \sigma_r^{dy}) \tag{4.6}$$

Die Regeln der Zustandsübergänge $\rightarrow_M$, $\rightarrow_A$, $\rightarrow_B$ und $\rightarrow_C$ von $[\![Serv_M]\!]$, $[\![Serv_A]\!]$, $[\![Serv_B]\!]$ und $[\![Serv_C]\!]$ bilden die Grundlage für die Komposition der Regeln zu einem Regelsystem für ein Service-orientiertes System.

Komposition zum Service-orientierten System

Nach Definition 3.7 im Grundlagenkapitel muss jede Prozedur oder Funktion p in einer benötigten Schnittstellenbeschreibung mit einer Prozedur oder Funktion p in einer zur Verfügung gestellten Schnittstelle verknüpft werden.

Der Klientenservice in Abbildung 4.21 enthält die Nutzungsschnittstelle R_M mit der Signatur $void\, a(int\, z)$. Diese Signatur kann mit der über die Angebotsschnittstelle I_A definierten Signatur $void\, a\, (int\, z)$ verknüpft werden. Analog können die Signaturen der Prozeduren der anderen Schnittstellen miteinander verknüpft werden. Aus den Services in Abbildung 4.21 kann so das Service-orientierte System in Abbildung 4.26 entstehen. Die gestrichelten Pfeile zeigen an, welche Services über welche Schnittstellen miteinander verbunden werden.

4.4.2. Komposition der Semantik der Services

Beispiel 4.5. Abbildung 4.26 zeigt ein Service-orientiertes System mit einem Klientenservice M und den Services A, B und C. In der Abbildung werden die Signaturen der Schnittstellen R_M, R_A, I_A. I_B und I_C von M, A, B und C definiert.

Der Klientenservice im Service-orientierten System in Abbildung 4.26 ist der Service M, der genau die Methode $main$ enthält. Im Rumpf von $main$ werden drei Funktionen aufgerufen, die der Klient M nicht selbst implementiert und die daher als die drei Signaturen in R_M wieder zu finden sind. Die Funktionen b und c müssen laut Schnittstellenbeschreibung R_M asynchrone Funktionen sein, und a muss eine synchrone Funktion sein. Der Klientenservice M stellt keine Angebotsschnittstellen zur Verfügung. Für die asynchronen Aufrufe von b und c werden die $futures$ f und g eingeführt, die jeweils an einen Funktion bei Aufruf gebunden werden, vgl. $f := b(x);$ und $g := c(x);$.

Der Service A in Abbildung 4.26 implementiert die synchrone Funktion a, wie angegeben in der Schnittstellenbeschreibung I_A. Im Programmpunkt q_{a2} und q_{a4} werden jeweils die asynchronen Prozeduren b und c aufgerufen, die Service A nicht selbst implementiert

(vgl. Schnittstellenbeschreibung R_A). Für den Aufruf der asynchronen Prozeduren b und c werden auch hier wie im Service M, zwei Programmvariablen f und g vom Typ $FUTURE$ benötigt. Die asynchronen Funktionsaufrufe werden jeweils einer Programmvariable zugewiesen, vgl. $q_{a2}: \ f := b(x);$ und $q_{a4}: \ g := c(x);$.

Die Services B und C hingegen bieten über die Angebotsschnittstellen I_B und I_C die Implementierungen der asynchronen Prozeduren a und b an. Beide Funktionen sind mit dem Schlüsselwort *async* gekennzeichnet und können nur asynchron aufgerufen werden. Service B und C benötigen keine weiteren Schnittstellen und haben keine Nutzungsschnittstellen.

Nachdem die Services M, A, B und C aus Beispiel 4.5 über die Nutzungs- und Angebotsschnittstellen das Service-orientierte System in Abbildung 4.26 bilden, können im nächsten Schritt die Regelsysteme der Zustandsübergangsrelationen über die operationalen Semantik eines jeden Services im Service-orientierten System miteinander vereinigt werden.

Die Komposition der Regelsysteme der Services M, A, B und C nach Def. 4.26, wobei M der Klient im Service-orientierten System ist, sei das Zustandsübergangssystem $[\![SoS]\!] = (Z_{SoS}, \rightarrow_{SoS}, Q_{SoS}, F_{Sos}, \mathcal{D}_{SoS})$ mit:

- $Z_{SoS} = PEX(DSFRAME_M) \cup PEX(DSFRAME_A) \cup PEX(DSFRAME_B) \cup PEX(DSFRAME_C)$,
- $\rightarrow_{SoS} = \rightarrow_M \cup \rightarrow_A \cup \rightarrow_B \cup \rightarrow_C$,
- $Q_{SoS} = \{(q_m, \sigma_0)\}$, wobei $(q_m, \sigma_0) \in DSFRAME_M$ Startzustand ist,
- $F_{SoS} = \{(q_f, \sigma)\}$, wobei $(q_f, \sigma) \in SFRAME$ Finalzustand, der letzte Punkt der main-Methode des Klienten M und σ ein beliebiger Speicher ist und
- $\mathcal{D}_{SoS} = \mathcal{D}_M \cup \mathcal{D}_A \cup \mathcal{D}_B \cup \mathcal{D}_C$ ist.

Um die Regeln der Komposition des Service-orientierten Systems ausführen zu können, müssen im nächsten Schritt die Platzhalter aufgelöst werden. Dazu wird die Abbildungsfunktion $\mathcal{D}_{SoS}$ genutzt, die sich aus der Definition der Abbildungsfunktionen $\mathcal{D}_A$, $\mathcal{D}_B$ und $\mathcal{D}_C$ der beteiligten Services ergibt.

In den Abbildungen 4.27 und 4.28 sind jeweils in der ersten Spalte die jeweiligen Zustandsübergangsrelationen mit Platzhalter, die farblich hinterlegt sind, zu sehen. In der jeweils zweiten Spalte sind die Zustandsübergangsrelationen nach Auflösung der Platzhalter auf konkrete Kaktuskeller-Elemente des aufgerufenen Service zu sehen. In den Abbildungen werden die Platzu

Die Services B und C müssen nicht weiter betrachtet werden, da beide Services keine Nutzungsschnittstellen haben und damit keine Servicegrenzen-übergreifende Funktionen oder Prozeduren aufrufen. Die Regeln der Zustandsübergangsrelation haben nur Platzhalter aus $DFRAME_B^{I_B}$ beziehungsweise $DFRAME_C^{I_C}$.

Nach Auflösung der Platzhalter bilden die Regeln der einzelnen Services das Regelsystem $\rightarrow_{SoS}$ des Service-orientierten Systems in Abbildung 4.26. Das Regelsystem ist in Abbildung 4.29 zu sehen.

Damit ergibt sich das Zustandsübergangssystem $[\![SoS]\!] = (Z_{SoS}, \rightarrow_{SoS}, Q_{SoS}, F_{Sos}, \mathcal{D}_{SoS})$ mit:

- Z_{SoS},
- $\rightarrow_{SoS}$ in Abbildung 4.29,
- $Q_{SoS} = \{(q_m, \sigma_0)\}$, wobei $(q_m, \sigma_0) \in DSFRAME_M$ Startzustand ist,
- $F_{SoS} = \{(q_f, \sigma)\}$, wobei $(q_f, \sigma) \in SFRAME$ Finalzustand, der letzte Punkt der *main*-Methode des Klienten M und σ ein beliebiger Speicher ist und

die Platzhalterfunktion $\mathcal{D}_{SoS}$ für das Service-orientierte System ist definiert durch:

$$\mathcal{D}_{SoS}(init_a^{I_A}, \sigma_i^{dy}) = (s_a, \sigma_i^{dy}) \tag{4.7}$$

$$\mathcal{D}_{SoS}(ret_a^{I_A}, \sigma_r^{dy}) = (r_a, \sigma_r^{dy}) \tag{4.8}$$

$$\mathcal{D}_{SoS}(init_b^{I_B}, \sigma_i^{dy}) = (s_b, \sigma_i^{dy}) \tag{4.9}$$

$$\mathcal{D}_{SoS}(ret_b^{I_B}, \sigma_r^{dy}) = (r_b, \sigma_r^{dy}) \tag{4.10}$$

$$\mathcal{D}_{SoS}(init_c^{I_C}, \sigma_i^{dy}) = (s_c, \sigma_i^{dy}) \tag{4.11}$$

$$\mathcal{D}_{SoS}(ret_c^{I_C}, \sigma_r^{dy}) = (r_c, \sigma_r^{dy}) \tag{4.12}$$

Mit dem Zustandsübergangssystem $[\![SoS]\!]$ kann das Verhalten des Service-orientierten Systems SoS bei unterschiedlichen Eingaben gezeigt werden.

Bemerkung 4.8. *Da der Speicher eines Zustands vom Eingabeparameter abhängt, kann die Menge der Zustände erst mit Ausführung des Service-orientierten Systems bestimmt werden. Die Menge der möglichen Zustände ist nicht endlich.*

4.4.3. Verhalten bei Ausführung

Im kommenden abschließenden Teil des ersten technischen Kapitels wird das Verhalten des in Abbildung 4.26 gezeigten Service-orientierten Systems mit Hilfe der operationalen Kaktuskeller-Semantik untersucht. Dabei wird das Verhalten bei Aufruf mit dem Eingabeparameter 1 ($x := 1$) und 2 ($x := 2$) gezeigt.

Ableitungsrelation bei Ausführung mit x:=1

Die Semantik des Service-orientierten Systems in Abbildung 4.26 mit dem Eingabeparameter 1 ($x := 1$) und mit dem Startzustand (q_m, σ_0) wird in Abbildung 4.30 als maximale Ableitungsfolge dargestellt. Dabei beschreibt die erste Spalte den korrespondierenden Kaktuskeller aus Abbildung 4.3, die zweite Spalte beschreibt die erreichbaren prozessalgebraischen Ausdrücke und die dritte Spalte, welche Regel aus $\rightarrow_{SoS}$ auf den prozessalgebraischen Ausdruck, und damit welche Regel der operationalen Semantik angewandt wurde.

Die Lauf des Zustandsübergangssystems des Service-orientierten Systems von Abbildung 4.26 in Abbildung 4.30 zeigt, dass bei der Eingabe von $[x := 1]$ das System vom Anfangszustand (q_m, σ_0) in den Finalzustand $(q_f, \sigma_3$ gelangt, wobei σ_3 den Speicherinhalt $[x := 1, f := (b, 1), g := (c, 2)])$ besitzt. Alle Kaktuskeller, die beim Aufruf der asynchronen Funktionen b und c entstehen, können wieder abgebaut werden, sodass als letztes Kaktuskeller-Element (q_f, σ_3) übrig bleibt. Die grafische Darstellung des Laufs wird in der Tabelle 4.3 und 4.4 gezeigt.

Grafische Darstellung der Ausführung mit x:=1

In Abbildung 4.3 sind die dazugehörigen Zustände in grafischer Darstellung als Kaktuskeller mit Kellerelementen aus der Menge $DSFRAME$ dargestellt. Die obersten Kaktuskeller-Elemente, auf denen Regeln angewandt werden können, werden gelb dargestellt.

Ableitungsrelation bei Ausführung mit x:=2

Die Semantik des Service-orientierten Systems in Abbildung 4.26 mit dem Eingabeparameter 2 ($x := 2$) und dem Startzustand (q_m, σ_0) wird in Abbildung 4.32 als maximale Ableitungsfolge dargestellt. Dabei beschreibt die erste Spalte den korrespondierenden Kaktuskeller aus Abbildung 4.5 und die zweite Spalte die erreichbaren Zustände (prozessalgebraische Zustände über Z_{SoS}) und die dritte Spalte, welche Anweisung am jeweili-

gen Programmpunkt und damit welche Regel der operationalen Semantik angewandt wurde.

Die Lauf des Zustandsübergangssystems des Service-orientierten Systems zeigt, dass das Service-orientierte System nicht vom Anfangszustand (q_m, σ_0) in den Finalzustand (q_f, σ) gelangen kann.

Der Lauf bei der Eingabe von 2 ($x := 2$) endet mit dem Ausdruck $(q_{a8}, \sigma_{11}).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy})$ mit dem Speicherinhalt $\sigma_{11} = [z := 2, x := 1, g := (c, 3)]), \sigma_3 = [x := 2, f := (b, 1), g := (c, 2)])$, $\sigma_3^{dy} = [b := 1]$ und dem Speicherinhalt $[c := 2, rval := 1])$ für den Speicher σ_5^{dy}. Dieser Ausdruck ist eine Normalform, da keine Regel aus $\rightarrow_{SoS}$ anwendbar ist. Da der Ausdruck nicht in der Menge der Finalzustände enthalten ist, handelt es damit um einen Deadlock. Dabei befindet sich die Ausführung am Programmpunkt q_{a8}, wo der zweite Aufruf der asynchronen Prozedur b synchronisiert werden soll. Als mögliche Kaktuskeller-Elemente zur Synchronisation stehen (r_b, σ_3^{dy}) und (r_c, σ_5^{dy}) nicht zur Verfügung, da diese Elemente an das Kaktuskeller-Element (q_f, σ_3) gebunden sind. Die grafische Darstellung des Laufs ist in den Tabellen 4.5 und 4.6 dargestellt. Dabei steht der Nummer des jeweiligen Kaktuskeller für den jeweiligen prozess-algebraischen Aufruf in Abbildung 4.32.

Grafische Darstellung der Ausführung mit x:=2

Zum besseren Verständnis sind den Tabellen 4.5 und 4.6 die dazugehörigen Zustände in grafischer Darstellung als Kaktuskeller dargestellt. Die obersten Kaktuskeller-Elemente, auf die die Regeln angewandt werden können, werden gelb dargestellt.

4.5. Zusammenfassung und Diskussion

Anhand zweier Beispiele wird gezeigt, dass das Verhalten eines Service-orientierten Systems von dem Eingabeparameter abhängen kann. In dem oben genannten Beispiel tritt bei dem Eingabeparameter 2 ein Deadlock auf. Bei Eingaben ungleich 2 tritt kein Deadlock auf, da nur bei Eingabe von 2 der Variable v im Service C am Programmpunkt q_{c1} der Wert 1 zugewiesen wird (durch Auswertung von $if(v == 2)$ zu wahr in Programmpunkt i_c). Durch diese Zuweisung wird im weiteren Programmverlauf im Service A die if-Anweisung am Programmpunkt q_{a7} zu $true$ ausgewertet und damit wird am Programmpunkt q_{a8} versucht, das den asynchronen Aufruf von b zu synchronisieren. Dies ist jedoch nicht möglich, da durch die anfängliche Auswertung der if-Anweisung in q_{a1} zu $falsch$ keine der if-Zweig nicht betreten wurde und damit kein asynchroner Aufruf mit dem future f verbunden ist.

Dieses Kapitel beschreibt auszugsweise die Beispielsprache $\mathcal{XYZ}$. Für die Beispielsprache wird eine operationale Kaktuskeller-Semantik eingeführt, die sowohl sequentielles, durch den „.“-Operator, als auch paralleles Verhalten, durch den „‖“-Operator, von monolithischen Systemen darstellbar ist. Durch Erweiterungen kann diese Semantik auch das Verhalten Service-orientierter Systeme zeigen. Anhand zweier Beispiele wird gezeigt, wie sich das Verhalten des Beispielsystem ändern kann und dass unter bestimmten Eingaben das Service-orientierte System in einem Deadlock enden kann.

Im kommenden Kapitel werden darauf aufbauend, Ansätze zur Deadlockanalyse in Service-orientierten Systemen unter Einhaltung der in der Einleitung getroffenen Annahmen, untersucht. Dabei wird der Abstraktionsprozess und zugrundeliegende Modelle diskutiert.

$(SP_r^{\rightarrow +})$	$\overline{(re_p, \sigma) \rightarrow (r_p^{I_A}, \sigma_r^{dy})}$	**Interface** $I_A\{void\ p();\}$ mit $void\, p(\ldots)\{\ldots, re_p : return;\}$
$(SPP_r^{\rightarrow +})$	$\overline{(re_p, \sigma) \rightarrow (r_p^{I_A}, \sigma_r^{dy}\vert_{rval}^{eval^\sigma(z)})}$	**Interface** $I_A\{int\ p();\}$ mit $int\, p(\ldots)\{\ldots, re_p : return\ z;\}$
$(AP_r^{\rightarrow +})$	$\overline{(re_p, \sigma) \rightarrow (r_p^{I_A}, \sigma_r^{dy}\vert_{p}^{eval^\sigma(p)})}$	**Interface** I_A $\{void\, \mathbf{async}\, (p()\vert p(int\, z_0, \ldots, int\, z_n));\}$ mit $void\, p(\ldots)\{\ldots, re_p : return;\}$
$(APP_r^{\rightarrow +})$	$\overline{(re_p, \sigma) \rightarrow (r_p^{I_A}, \sigma_r^{dy}\vert_{p,rval}^{eval^\sigma(p),eval^\sigma(z)})}$	**Interface** I_A $\{int\, \mathbf{async}\, (p()\vert p(int\, z_0, \ldots, int\, z_n));\}$ mit $int\, p(\ldots)\{\ldots, re_p : return\ z;\}$

Wobei $r_p^{I_A}$ zusätzliche Rückkehr-Programmpunkt, re_p der letzte Programmpunkt der Prozedur p ist ($return$-Anweisung), und σ_r^{dy} ein Rückkehr-Dummy-Speicher, σ ein regulärer Speicher sei. I_A sei die angebotene Schnittstelle von Service A, der p implementiert.

Abbildung 4.20.: Regeln der Operationalen Kaktuskeller-Semantik für Services für die Rückkehr.

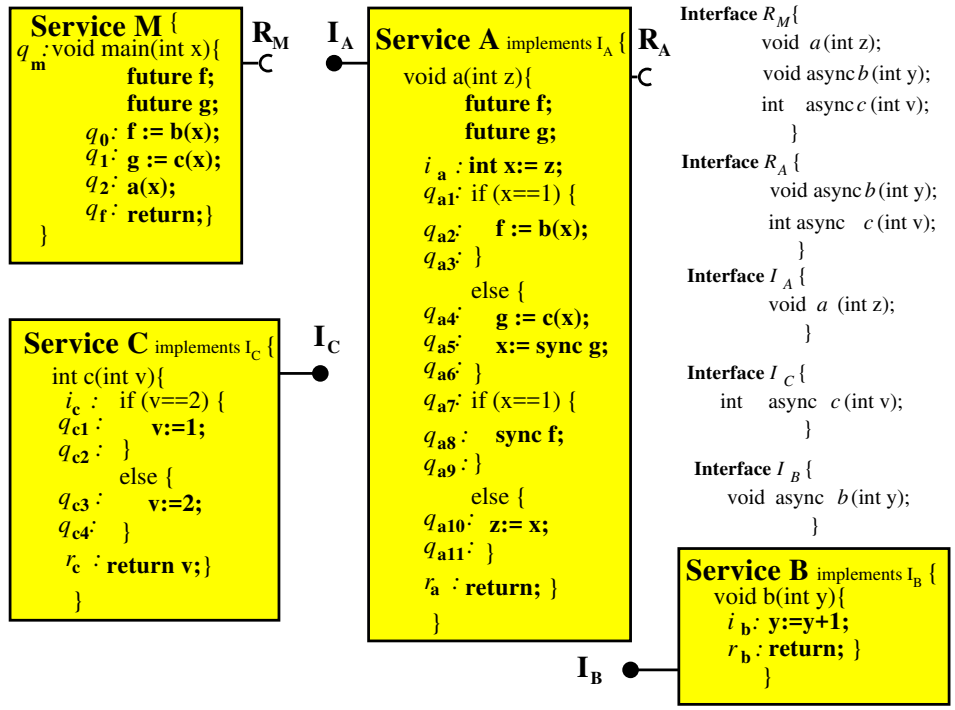


Abbildung 4.21.: Services M, A, B und C mit den Signaturen für deren Schnittstellen R_M, R_A, I_A, I_B und I_C.

Quellcode von Service M, Fig. 4.21	Zustandsübergangsregeln $\rightarrow_M$
q_M : *void main(int x){* *future f;* *future g;* q_0 : **f:=b(x)**; q_1 : **g:=c(x)**; q_2 : **a(x)**; q_f : **return**; }	$\frac{}{(q_m,\sigma_0)\rightarrow(q_0,\sigma\vert_x^{eval^{\sigma_0}(x)})}$, $\frac{}{(q_0,\sigma)\rightarrow(q_1,\sigma'\vert_f^{(b,1)})\Vert(init_b^{I_{S_b}},\sigma_b^{dy}\vert_{b,x_0}^{1,eval^{\sigma}(x)})}$, $\frac{}{(q_1,\sigma')\rightarrow(q_2,\sigma''\vert_g^{(c,1)})\Vert(init_c^{I_{S_c}},\sigma_c^{dy}\vert_{c,x_0}^{1,eval^{\sigma}_A(x)})}$, $\frac{}{(q_2,\sigma'')\rightarrow(init_a^{I_{S_a}},\sigma_a^{dy}).(q_f,\sigma'')}$, $\frac{\sigma''(f)=(b,\sigma_b'^{dy}(b))}{(q_f,\sigma'')\Vert(ret_b^{I_{S_b}},\sigma_b'^{dy})\rightarrow(q_f,\sigma'')}$, $\frac{\sigma''(g)=(c,\sigma_c'^{dy}(c))}{(q_f,\sigma'')\Vert(ret_c^{I_{S_c}},\sigma_c'^{dy})\rightarrow(q_f,\sigma'')}$, $\frac{}{(ret_a^{I_{S_a}},\sigma_a'^{dy}).(q_f,\sigma'')\rightarrow(q_f,\sigma'')}$

Abbildung 4.22.: Das Regelsystem $\rightarrow_{M_a}$ für den Service M aus Abbildung 4.21. Die Platzhalter $(init_x^{I_{S_x}},\sigma_x^{dy})$ und $(ret_x^{I_{S_x}},\sigma_x^{dy})$ mit $x \in \{a,b,c\}$ stehen für die jeweiligen benötigten Prozeduren wie in der Schnittstelle R_M beschrieben.

Quellcode von Service A, Fig. 4.21	Regelsystem $\rightarrow_A$
$void\ a(int\ z)\{$ $\quad future\ f;$ $\quad future\ g;$ i_a : **int x:=z**; q_{a1} : **if(x==1){** q_{a2} : **f:=b(x)**; q_{a3} :**}** **else{** q_{a4} : **g:=c(x)**; q_{a5} : **x:=sync g**; q_{a6} :**}** q_{a7} : **if(x==1)** q_{a8} : **sync f**; q_{a9} :**}** $else\{$ q_{a10} : **z:=x**; q_{a11} :**}** re_a : **return**;$\}$	$\frac{}{(s_a^{I_A},\sigma_a^{dy})\rightarrow(i_a,\sigma\vert_z^{eval^{\sigma_a^{dy}}(x_0)})}$, $\frac{}{(i_a,\sigma)\rightarrow(q_{a1},\sigma'\vert_x^{eval^{\sigma}(z)})}$, $\frac{eval^{\sigma'}(x==1)=true}{(q_{a1},\sigma')\rightarrow(q_{a2},\sigma')}$, $\frac{eval^{\sigma'}(x==1)=false}{(q_{a1},\sigma')\rightarrow(q_{a4},\sigma')}$, $\frac{}{(q_{a2},\sigma')\rightarrow(init_b^{I_{S_b}},\sigma_b^{dy}\vert_{b,a_1}^{1,eval^{\sigma}(x)})\|(q_{a3},\sigma''\vert_f^{(b,1)})}$, $\frac{}{(q_{a3},\sigma'')\rightarrow(q_{a7},\sigma'')}$, $\frac{}{(q_{a4},\sigma')\rightarrow(init_c^{I_{S_c}},\sigma_c^{dy}\vert_{c,x_0}^{1,eval^{\sigma}(x)})\|(q_{a5},\sigma'''\vert_g^{(c,1)})}$, $\frac{\sigma'''(g)=(c,\sigma_c^{dy}(c))}{(q_{a5},\sigma''')\|(ret_c^{I_{S_c}},\sigma_c'^{dy})\rightarrow(q_{a6},\sigma'')}$, $\frac{}{(q_{a6},\sigma'')\rightarrow(q_{a7},\sigma'')}$, $\frac{eval^{\sigma''}(x==1)=true}{(q_{a7},\sigma'')\rightarrow(q_{a8},\sigma)}$, $\frac{eval^{\sigma''}(x==1)=false}{(q_{a7},\sigma'')\rightarrow(q_{a10},\sigma'')}$, $\frac{\sigma''(f)=(b,\sigma_b^{dy}(b))}{(q_{a8},\sigma'')\|(ret_b^{I_{S_b}},\sigma_b^{dy})\rightarrow(q_{a9},\sigma'')}$, $\frac{}{(q_{a9},\sigma'')\rightarrow(re_a,\sigma''')}$, $\frac{}{(q_{a10},\sigma'')\rightarrow(q_{a11},\sigma'''\vert_z^{eval^{\sigma}(x)})}$, $\frac{}{(q_{a11},\sigma''')\rightarrow(re_a,\sigma''')}$, $\frac{\sigma(g)=(c,\sigma_c'^{dy}(c))}{(re_a,\sigma''')\|(ret_c^{I_{S_c}},\sigma_c'^{dy})\rightarrow(re_a,\sigma''')}$, $\frac{\sigma(f)=(b,\sigma_b'^{dy}(b))}{(re_a,\sigma''')\|(ret_b^{I_{S_b}},\sigma_b'^{dy})\rightarrow(re_a,\sigma''')}$, $\frac{}{(re_a,\sigma''')\rightarrow(r_a^{I_A},\sigma_b^{dy})}$

Abbildung 4.23.: Das Regelsystem $\rightarrow_a$ der Zustandsübergänge für den Service A aus Abbildung 4.21.

Quellcode von Service B, Fig. 4.21	Regelsystem $\rightarrow_B$
$void\ b(int\ y)\{$ i_b : **y:=y+1**; re_b : **return**;$\}$	$\frac{}{(s_b^{I_B},\sigma_b^{dy})\rightarrow(i_b,\sigma\vert_y^{eval^{\sigma_b^{dy}}(x_0)})}$, $\frac{}{(i_b,\sigma)\rightarrow(re_b,\sigma'\vert_y^{eval^{\sigma}(y+1)})}$, $\frac{}{(re_b,\sigma')\rightarrow(r_b^{I_B},\sigma_b'^{dy})}$

Abbildung 4.24.: Die Regeln $\rightarrow_b$ der Zustandsübergänge für den Service B aus Abbildung 4.21

Quellcode von Service C, Fig. 4.21	Regelsystem
$int\ c(int\ v)\{$ i_c : **if(v==2){** q_{c1} : **v:=1;** q_{c2} :**}** **else{** q_{c3} : **v:=2;** q_{c4} : **}** re_c : **return v;}**	$\frac{}{(s_c^{IC},\sigma_c^{dy})\rightarrow(i_c,\sigma|_{v,c}^{eval^{\sigma_c^{dy}}(x_0),eval^{\sigma_c^{dy}}(c)})}$, $\frac{eval^{\sigma}(v==2)=true}{(i_c,\sigma)\rightarrow(q_{c1},\sigma)}$, $\frac{eval_B^{\sigma}(v==2)=false}{(i_c,\sigma)\rightarrow(q_{c3},\sigma)}$, $\frac{}{(q_{c1},\sigma)\rightarrow(q_{c2},\sigma'|_v^{eval^{\sigma}(1)})}$, $\frac{}{(q_{c2},\sigma')\rightarrow(re_c,\sigma'')}$, $\frac{}{(q_{c3},\sigma)\rightarrow(q_{c4},\sigma'|_v^{eval^{\sigma}(2)})}$, $\frac{}{(q_{c4},\sigma')\rightarrow(re_c,\sigma'')}$, $\frac{}{(re_c,\sigma'')\rightarrow(r_c^{IC},\sigma_c'^{dy}|_{c,rval}^{eval^{\sigma''}(c),eval^{\sigma''}(v)})}$

Abbildung 4.25.: Die Regeln $\rightarrow_c$ der Zustandsübergänge für den Service C aus Abbildung 4.21

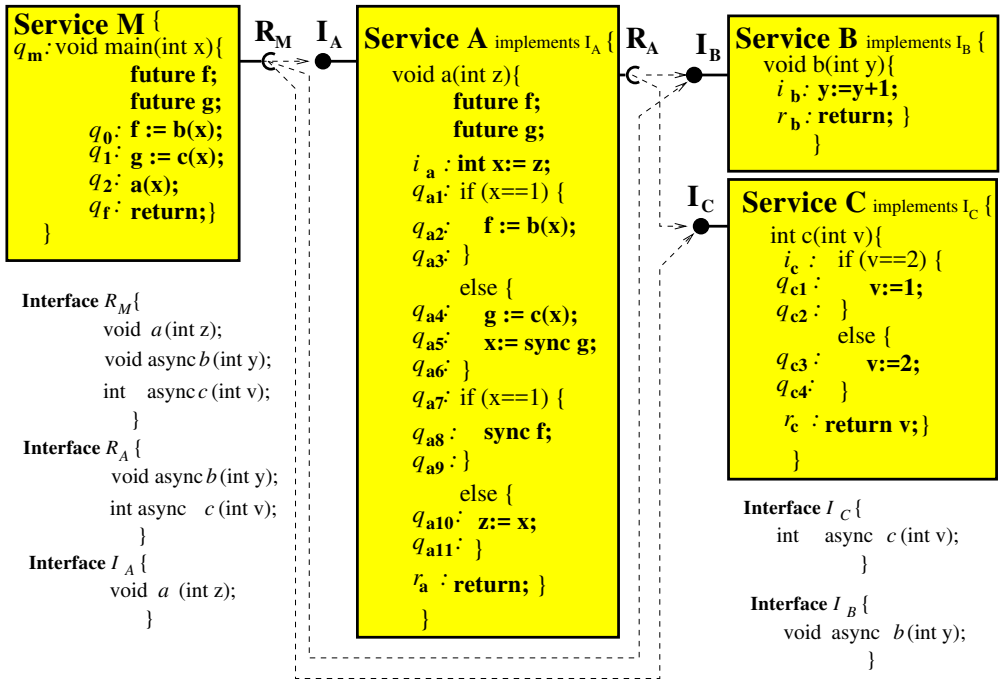


Abbildung 4.26.: Ein Service-orientiertes System bestehend aus den Diensten M, A, B und C mit den Signaturen für die Schnittstellen R_M, R_A, I_A, I_B und I_C.

Regelsystem von Service M mit Platzhaltern	Regelsystem von Service M nach Auflösung der Platzhalter durch die Platzhalterfunktionen $\mathcal{D}_A$, $\mathcal{D}_B$ und $\mathcal{D}_C$
$\frac{}{(q_m,\sigma_0)\to(q_0,\sigma\vert_x^{eval^{\sigma_0}(x)})}$, $\frac{}{(q_0,\sigma)\to(q_1,\sigma'\vert_f^{(b,1)})\Vert\ (init_b^{I_{S_b}},\sigma_b^{dy}\vert_{b,x_0}^{1,eval^{\sigma}(x)})}$, $\frac{}{(q_1,\sigma')\to(q_2,\sigma''\vert_g^{(c,1)})\Vert\ (init_c^{I_{S_c}},\sigma_c^{dy}\vert_{c,x_0}^{1,eval_A^{\sigma}(x)})}$, $\frac{}{(q_2,\sigma'')\to\ (init_a^{I_{S_a}},\sigma_a^{dy}).(q_f,\sigma'')}$, $\frac{\sigma''(f)=(b,\sigma_b'^{dy}(b))}{(q_f,\sigma'')\Vert\ (ret_b^{I_{S_b}},\sigma_b'^{dy})\to(q_f,\sigma'')}$, $\frac{\sigma''(g)=(c,\sigma_c'^{dy}(c))}{(q_f,\sigma'')\Vert\ (ret_c^{I_{S_c}},\sigma_c'^{dy})\to(q_f,\sigma'')}$, $\frac{}{(ret_a^{I_{S_a}},\sigma_a'^{dy}).(q_f,\sigma'')\to(q_f,\sigma'')}$	$\frac{}{(q_0,\sigma)\to(q_1,\sigma'\vert_f^{(b,1)})\Vert\ (s_b^{I_{S_b}},\sigma_b^{dy}\vert_{b,x_0}^{1,eval^{\sigma}(x)})}$, $\frac{}{(q_1,\sigma')\to(q_2,\sigma''\vert_g^{(c,1)})\Vert\ (s_c^{I_{S_c}},\sigma_c^{dy}\vert_{c,x_0}^{1,eval_A^{\sigma}(x)})}$, $\frac{}{(q_2,\sigma'')\to\ (s_a^{I_{A_a}},\sigma_a^{dy}).(q_f,\sigma'')}$, $\frac{\sigma''(f)=(b,\sigma_b'^{dy}(b))}{(q_f,\sigma'')\Vert\ (r_b^{I_{S_b}},\sigma_b'^{dy})\to(q_f,\sigma'')}$, $\frac{\sigma''(g)=(c,\sigma_c'^{dy}(c))}{(q_f,\sigma'')\Vert\ (r_c^{I_{S_c}},\sigma_c'^{dy})\to(q_f,\sigma'')}$, $\frac{}{(r_a^{I_{A_a}},\sigma_a'^{dy}).(q_f,\sigma'')\to(q_f,\sigma'')}$
Auflösung der Platzhalter von Service M für Service C mit $\mathcal{D}_C$: $(init_c^{I_C},\sigma_c^{dy})\mapsto(s_c^{I_C},\sigma_c^{dy})$ $(ret_c^{I_C},\sigma_c^{dy})\mapsto(r_c^{I_C},\sigma_c'^{dy})$	Auflösung der Platzhalterfunktion von Service M für Service B mit $\mathcal{D}_B$: $(init_b^{I_B},\sigma^{dy})\mapsto(s_b^{I_B},\sigma^{dy})$ $(ret_b^{I_B},\sigma^{dy})(r_b^{I_B},\sigma^{dy})$
Auflösung der Platzhalter von Service M für Service A mit $\mathcal{D}_A$: $(init_a^{I_A},\sigma^{dy})\mapsto(s_a^{I_A},\sigma^{dy})$ $(ret_a^{I_A},\sigma^{dy})\mapsto(r_a,\sigma^{dy})$	Die Platzhalterfunktionen werden bei Bindung an den jeweiligen Service über die Angebotsschnittstellen I_A, I_B und I_C zur Verfügung gestellt.

Abbildung 4.27.: Auflösung der Platzhalter von Service M.

Regelsystem von Service A mit Platzhaltern	Regelsystem von Service A nach Auflösung der Platzhalter durch die Platzhalterfunktionen $\mathcal{D}_B$ und $\mathcal{D}_C$
$\frac{}{(s_a^{I_A},\sigma_a^{dy})\to(i_a,\sigma|_z^{eval^{\sigma_a^{dy}}(x_0)})}$, $\frac{}{(i_a,\sigma)\to(q_{a1},\sigma'|_x^{eval^{\sigma}(z)})}$, $\frac{eval^{\sigma'}(x==1)=true}{(q_{a1},\sigma')\to(q_{a2},\sigma')}$, $\frac{eval^{\sigma'}(x==1)=false}{(q_{a1},\sigma')\to(q_{a4},\sigma')}$, $\frac{}{(q_{a2},\sigma')\to (init_b^{I_{S_b}},\sigma_b^{dy}|_{b,a_1}^{1,eval^{\sigma}(x)})\ \|(q_{a3},\sigma''|_f^{(b,1)})}$, $\frac{}{(q_{a3},\sigma'')\to(q_{a7},\sigma'')}$, $\frac{}{(q_{a4},\sigma')\to (init_c^{I_{S_c}},\sigma_c^{dy}|_{c,x_0}^{1,eval^{\sigma}(x)})\ \|(q_{a_5},\sigma'''|_g^{(c,1)})}$, $\frac{\sigma'''(g)=(c,\sigma_c^{dy}(c))}{(q_{a5},\sigma'''\|\ (ret_c^{I_{S_c}},\sigma_c'^{dy})\to(q_{a6},\sigma'')}$, $\frac{}{(q_{a6},\sigma'')\to(q_{a7},\sigma'')}$, $\frac{eval^{\sigma''}(x==1)=true}{(q_{a7},\sigma'')\to(q_{a8},\sigma)}$, $\frac{eval^{\sigma''}(x==1)=false}{(q_{a7},\sigma'')\to(q_{a10},\sigma'')}$, $\frac{\sigma''(f)=(b,\sigma_b^{dy}(b))}{(q_{a8},\sigma''\|\ (ret_b^{I_{S_b}},\sigma_b^{dy})\to(q_{a9},\sigma'')}$, $\frac{}{(q_{a9},\sigma'')\to(re_a,\sigma''')}$, $\frac{}{(q_{a10},\sigma'')\to(q_{a11},\sigma'''|_z^{eval^{\sigma}(x)})}$, $\frac{}{(q_{a11},\sigma''')\to(re_a,\sigma''')}$, $\frac{\sigma(g)=(c,\sigma_c'^{dy}(c))}{(re_a,\sigma'''\|\ (ret_c^{I_{S_c}},\sigma_c'^{dy})\to(re_a,\sigma''')}$, $\frac{\sigma(f)=(b,\sigma_b'^{dy}(b))}{(re_a,\sigma'''\|\ (ret_b^{I_{S_b}},\sigma_b'^{dy})\to(re_a,\sigma'''}$, $r\frac{}{(re_a,\sigma''')\to(r_a^{I_A},\sigma_b^{dy}),\sigma'''}$	$\frac{}{(s_a^{I_A},\sigma_a^{dy})\to(i_a,\sigma|_z^{eval^{\sigma_a^{dy}}(x_0)})}$, $\frac{}{(i_a,\sigma)\to(q_{a1},\sigma'|_x^{eval^{\sigma}(z)})}$, $\frac{eval^{\sigma'}(x==1)=true}{(q_{a1},\sigma')\to(q_{a2},\sigma')}$, $\frac{eval^{\sigma'}(x==1)=false}{(q_{a1},\sigma')\to(q_{a4},\sigma')}$, $\frac{}{(q_{a2},\sigma')\to (s_b^{I_{B_b}},\sigma_b^{dy}|_{b,a_1}^{1,eval^{\sigma}(x)})\ \|(q_{a3},\sigma''|_f^{(b,1)})}$, $\frac{}{(q_{a3},\sigma'')\to(q_{a7},\sigma'')}$, $\frac{}{(q_{a4},\sigma')\to (s_c^{I_{C_c}},\sigma_c^{dy}|_{c,x_0}^{1,eval^{\sigma}(x)})\ \|(q_{a_5},\sigma'''|_g^{(c,1)})}$, $\frac{\sigma'''(g)=(c,\sigma_c^{dy}(c))}{(q_{a5},\sigma'''\|\ (r_c^{I_{C_c}},\sigma_c'^{dy})\to(q_{a6},\sigma'')}$, $\frac{}{(q_{a6},\sigma'')\to(q_{a7},\sigma'')}$, $\frac{eval^{\sigma''}(x==1)=true}{(q_{a7},\sigma'')\to(q_{a8},\sigma)}$, $\frac{eval^{\sigma''}(x==1)=false}{(q_{a7},\sigma'')\to(q_{a10},\sigma'')}$, $\frac{\sigma''(f)=(b,\sigma_b^{dy}(b))}{(q_{a8},\sigma''\|\ (r_b^{I_{B_b}},\sigma_b^{dy})\to(q_{a9},\sigma'')}$, $\frac{}{(q_{a9},\sigma'')\to(re_a,\sigma''')}$, $\frac{}{(q_{a10},\sigma'')\to(q_{a11},\sigma'''|_z^{eval^{\sigma}(x)})}$, $\frac{}{(q_{a11},\sigma''')\to(re_a,\sigma''')}$, $\frac{\sigma(g)=(c,\sigma_c'^{dy}(c))}{(re_a,\sigma'''\|\ (r_c^{I_{C_c}},\sigma_c'^{dy})\to(re_a,\sigma''')}$, $\frac{\sigma(f)=(b,\sigma_b'^{dy}(b))}{(re_a,\sigma'''\|\ (r_b^{I_{B_b}},\sigma_b'^{dy})\to(re_a,\sigma'''}$, $r\frac{}{(re_a,\sigma''')\to(r_a^{I_A},\sigma_b^{dy}),\sigma'''}$
Auflösung der Platzhalter von Service A für Service B mit $\mathcal{D}_B$: $(init_c^{I_S},\sigma_c^{dy})\mapsto(s_c^{I_C},\sigma_c^{dy})$ $(ret_c^{I_S},\sigma_c^{dy})\mapsto(r_c^{I_C},\sigma_c'^{dy})$	Auflösung der Platzhalterfunktion von Service A für Service C mit $\mathcal{D}_C$: $(init_b^{I_S},\sigma^{dy})\mapsto(s_b^{I_C},\sigma^{dy})$ $(ret_b^{I_S},\sigma^{dy})\mapsto(r_b^{I_C},\sigma^{dy})$
	Die Platzhalterfunktionen werden bei Bindung an den jeweiligen Service über die Angebotsschnittstellen I_B und I_C zur Verfügung gestellt.

Abbildung 4.28.: Auflösung der Platzhalter von Service A.

Service M	Service A
$\frac{}{(q_m,\sigma_0)\to(q_0,\sigma\vert_x^{eval^{\sigma_0}(x)})}$, $\frac{}{(q_0,\sigma)\to(q_1,\sigma'\vert_f^{(b,1)})\Vert(s_b^{I_{S_b}},\sigma_b^{dy}\vert_{b,x_0}^{1,eval^{\sigma}(x)})}$, $\frac{}{(q_1,\sigma')\to(q_2,\sigma''\vert_g^{(c,1)})\Vert(s_c^{I_{S_c}},\sigma_c^{dy}\vert_{c,x_0}^{1,eval^{\sigma}_A(x)})}$, $\frac{}{(q_2,\sigma'')\to(s_a^{I_{A_a}},\sigma_a^{dy}).(q_f,\sigma'')}$, $\frac{\sigma''(f)=(b,\sigma_b'^{dy}(b))}{(q_f,\sigma'')\Vert(r_b^{I_{S_b}},\sigma_b'^{dy})\to(q_f,\sigma'')}$, $\frac{\sigma''(g)=(c,\sigma_c'^{dy}(c))}{(q_f,\sigma'')\Vert(r_c^{I_{S_c}},\sigma_c'^{dy})\to(q_f,\sigma'')}$, $\frac{}{(r_a^{I_{A_a}},\sigma_a'^{dy}).(q_f,\sigma'')\to(q_f,\sigma'')}$	$\frac{}{(s_a^{I_A},\sigma_a^{dy})\to(i_a,\sigma\vert_z^{eval^{\sigma_a^{dy}}(x_0)})}$, $\frac{}{(i_a,\sigma)\to(q_{a1},\sigma'\vert_x^{eval^{\sigma}(z)})}$, $\frac{eval^{\sigma'}(x==1)=true}{(q_{a1},\sigma')\to(q_{a2},\sigma')}$, $\frac{eval^{\sigma'}(x==1)=false}{(q_{a1},\sigma')\to(q_{a4},\sigma')}$, $\frac{}{(q_{a2},\sigma')\to(s_b^{I_{B_b}},\sigma_b^{dy}\vert_{b,a_1}^{1,eval^{\sigma}(x)})\Vert(q_{a3},\sigma''\vert_f^{(b,1)})}$, $\frac{}{(q_{a3},\sigma'')\to(q_{a7},\sigma'')}$, $\frac{}{(q_{a4},\sigma')\to(s_c^{I_{C_c}},\sigma_c^{dy}\vert_{c,x_0}^{1,eval^{\sigma}(x)})\Vert(q_{a_5},\sigma'''\vert_g^{(c,1)})}$, $\frac{\sigma'''(g)=(c,\sigma_c^{dy}(c))}{(q_{a5},\sigma''')\Vert(r_c^{I_{C_c}},\sigma_c'^{dy})\to(q_{a6},\sigma'')}$, $\frac{}{(q_{a6},\sigma'')\to(q_{a7},\sigma'')}$, $\frac{eval^{\sigma''}(x==1)=true}{(q_{a7},\sigma'')\to(q_{a8},\sigma)}$, $\frac{eval^{\sigma''}(x==1)=false}{(q_{a7},\sigma'')\to(q_{a10},\sigma'')}$, $\frac{\sigma''(f)=(b,\sigma_b^{dy}(b))}{(q_{a8},\sigma'')\Vert(r_b^{I_{B_b}},\sigma_b^{dy})\to(q_{a9},\sigma'')}$, $\frac{}{(q_{a9},\sigma'')\to(re_a,\sigma''')}$, $\frac{}{(q_{a10},\sigma'')\to(q_{a11},\sigma'''\vert_z^{eval^{\sigma}(x)})}$, $\frac{}{(q_{a11},\sigma''')\to(re_a,\sigma''')}$, $\frac{\sigma(g)=(c,\sigma_c'^{dy}(c))}{(re_a,\sigma''')\Vert(r_c^{I_{C_c}},\sigma_c'^{dy})\to(re_a,\sigma''')}$, $\frac{\sigma(f)=(b,\sigma_b'^{dy}(b))}{(re_a,\sigma''')\Vert(r_b^{I_{B_b}},\sigma_b'^{dy})\to(re_a,\sigma'''}$, $r\frac{}{(re_a,\sigma''')\to(r_a^{I_A},\sigma_b^{dy}),\sigma'''}$
Service B	**Service C**
$\frac{}{(s_b^{I_B},\sigma_b^{dy})\to(i_b,\sigma\vert_y^{eval^{\sigma_b^{dy}}(x_0)})}$, $\frac{}{(i_b,\sigma)\to(re_b,\sigma'\vert_y^{eval^{\sigma}(y+1)})}$, $\frac{}{(re_b,\sigma')\to(r_b^{I_B},\sigma_b'^{dy})}$	$\frac{}{(s_c^{I_C},\sigma_c^{dy})\to(i_c,\sigma\vert_{v,c}^{eval^{\sigma_c^{dy}}(x_0),eval^{\sigma_c^{dy}}(c)})}$, $\frac{eval^{\sigma}(v==2)=true}{(i_c,\sigma)\to(q_{c1},\sigma)}$, $\frac{eval_B^{\sigma}(v==2)=false}{(i_c,\sigma)\to(q_{c3},\sigma)}$, $\frac{}{(q_{c1},\sigma)\to(q_{c2},\sigma'\vert_v^{eval^{\sigma}(1)})}$, $\frac{}{(q_{c2},\sigma')\to(re_c,\sigma'')}$, $\frac{}{(q_{c3},\sigma)\to(q_{c4},\sigma'\vert_v^{eval^{\sigma}(2)})}$, $\frac{}{(q_{c4},\sigma')\to(re_c,\sigma'')}$, $\frac{}{(re_c,\sigma'')\to(r_c^{I_C},\sigma_c'^{dy}\vert_{c,rval}^{eval^{\sigma''}(c),eval^{\sigma''}(v)})}$

Abbildung 4.29.: Das Regelsystem $\to_{sos}$ des Service-orientierten Systems aus Abbildung 4.26.

Kaktus-keller in Abb. 4.3	Ableitungsrelation	Regel aus $\rightarrow_{SAB}$
(0)	$\underline{(q_m, \sigma_0)} \Rightarrow$	$(MAIN_1)$
(1)	$\underline{(q_0, \sigma_1)} \Rightarrow$	$(PA_1^{\rightarrow +})$
(2)	$\underline{(q_1, \sigma_2)} \parallel (s_b, \sigma_1^{dy}) \Rightarrow$	$(PA_1^{\rightarrow +})$
(3)	$(q_2, \sigma_3) \parallel (s_c, \sigma_2^{dy}) \parallel \underline{(s_b, \sigma_1^{dy})} \Rightarrow$	$(APP_i^{\rightarrow +})$
(4)	$(q_2, \sigma_3) \parallel (s_c, \sigma_2^{dy}) \parallel \underline{(i_b, \sigma_4)} \Rightarrow$	(ZU)
(5)	$(q_2, \sigma_3) \parallel (s_c, \sigma_2^{dy}) \parallel \underline{(re_b, \sigma_5)} \Rightarrow$	$AP_r^{\rightarrow +})$
(6)	$(q_2, \sigma_3) \parallel \underline{(s_c, \sigma_2^{dy})} \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	$(APP_r^{\rightarrow +})$
(7)	$\underline{(q_2, \sigma_3)} \parallel (i_c, \sigma_6) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	$(PS_1^{\rightarrow +})$
(8)	$\underline{(s_a, \sigma_4^{dy})}.(q_f, \sigma_3) \parallel (i_c, \sigma_6) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	$(SPP_i^{\rightarrow +})$
(9)	$(i_a, \sigma_7).(q_f, \sigma_3]) \parallel \underline{(i_c, \sigma_6)} \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	(IF_{false})
(10)	$(i_a, \sigma_7).(q_f, \sigma_3) \parallel \underline{(q_{c3}, \sigma_6)} \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	(ZU)
(11)	$\underline{(i_a, \sigma_7)}.(q_f, \sigma_3) \parallel (q_{c4}, \sigma_8) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	(ZU)
(12)	$(q_{a1}, \sigma_9).(q_f, \sigma_3) \parallel \underline{(q_{c4}, \sigma_8)} \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	(IF_{false_r})
(13)	$(q_{a1}, \sigma_9).(q_f, \sigma_3) \parallel \underline{(re_c, \sigma_8)} \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	$(APP_r^{\rightarrow +})$
(14)	$\underline{(q_{a1}, \sigma_9)}.(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy}) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	(IF_{true})
(15)	$\underline{(q_{a2}, \sigma_9)}.(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy}) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	$(PA_1^{\rightarrow +})$
(16)	$(\underline{(q_{a3}, \sigma_{10})} \parallel (s_b, \sigma_5^{dy})).(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy}) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	(IF_{true_r})
(17)	$((q_{a7}, \sigma_{10}) \parallel (s_b, \sigma_5^{dy})).(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy}) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	$(APP_i^{\rightarrow +})$
(18)	$((q_{a7}, \sigma_{10}) \parallel \underline{(i_b, \sigma_{11})}.(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy}) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	(ZU)
(19)	$((q_{a7}, \sigma_{10}) \parallel \underline{(re_b, \sigma_{12})}).(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy}) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	$(APP_r^{\rightarrow +})$
(20)	$(\underline{(q_{a7}, \sigma_{10})} \parallel (r_b, \sigma_6^{dy})).(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy}) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	(IF_{true})
(21)	$(\underline{(q_{a8}, \sigma_{10})} \parallel (r_b, \sigma_6^{dy})).(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy}) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	$(PA_{sync}^{\rightarrow +})$
(22)	$(\underline{(q_{a9}, \sigma_{10})}.(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy}) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	(IF_{true_r})
(23)	$(\underline{(re_a, \sigma_{10})}.(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy}) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	$(SP_r^{\rightarrow +})$
(24)	$(\underline{(r_a, \sigma_7^{dy})}.(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy}) \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	$(PS_1^{\rightarrow +})$
(25)	$(\underline{(q_f, \sigma_3) \parallel (r_c, \sigma_4^{dy})} \parallel (r_b, \sigma_3^{dy}) \Rightarrow$	$(PA_1^{\rightarrow +})$
(26)	$\underline{(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy})} \Rightarrow$	$(PA_0^{\rightarrow +})$
(27)	(q_f, σ_3)	*Finalzustand*

Abbildung 4.30.: Die Semantik des Service-orientierten Systems (Lauf der operationalen Semantik) mit der Eingabe main(1) aus Abbildung 4.26, $x := 1$ als Ableitungsfolge.

Speicher	Inhalt	Dummy-Speicher	Inhalt
σ_0	$[]$	σ_1^{dy}	$[b := 1, x_0 := 1]$
σ_1	$[x := 1]$	σ_2^{dy}	$[c := 2, x_0 := 1]$
σ_2	$[x := 1, f := (b, 1)]$	σ_3^{dy}	$[b := 1]$
σ_3	$[x := 1, f := (b, 1), g := (c, 2)]$	σ_4^{dy}	$[c := 2, rval := 2]$
σ_4	$[b := 1, y := 1]$	σ_5^{dy}	$[b := 3, x_0 := 1]$
σ_5	$[b := 1, y := 2]$	σ_6^{dy}	$[b := 3]$
σ_6	$[c := 2, v := 1]$	σ_7^{dy}	$[]$
σ_7	$[z := 1]$		
σ_8	$[c := 2, v := 2]$		
σ_9	$[z := 1, x := 1]$		
σ_{10}	$[z := 1, x := 1, f = (b, 3)]$		
σ_{11}	$[b := 3, y := 1]$		
σ_{12}	$[b := 3, y := 2]$		

Abbildung 4.31.: Der Inhalt der Speicher und Dummy-Speicher aus Abbildung 4.30.

Kaktus-keller in Abb. 4.5	Ableitungsrelation	Regel aus $\rightarrow_{SAB}$
(0)	$\underline{(q_m, \sigma_0)} \Rightarrow$	$(MAIN_1)$
(1)	$\underline{(q_0, \sigma_1)} \Rightarrow$	$(PA_1^{\rightarrow +})$
(2)	$\underline{(q_1, \sigma_2)} \parallel (s_b, \sigma_1^{dy}) \Rightarrow$	$(PA_1^{\rightarrow +})$
(3)	$(q_2, \sigma_3) \parallel \underline{(s_b, \sigma_1^{dy})} \parallel (s_c, \sigma_2^{dy}[c := 2, x_0 := 2]) \Rightarrow$	$(APP_i^{\rightarrow +})$
(4)	$(q_2, \sigma_3) \parallel \underline{(i_b, \sigma_4)} \parallel (s_c, \sigma_2^{dy}) \Rightarrow$	(ZU)
(5)	$(q_2, \sigma_3) \parallel \underline{(re_b, \sigma_5)} \parallel (s_c, \sigma_2^{dy}) \Rightarrow$	$(AP_r^{\rightarrow +})$
(6)	$(q_2, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel \underline{(s_c, \sigma_2^{dy})} \Rightarrow$	$(APP_r^{\rightarrow +})$
(7)	$\underline{(q_2, \sigma_3)} \parallel (r_b, \sigma_3^{dy}) \parallel (i_c, \sigma_6) \Rightarrow$	$(PS_1^{\rightarrow +})$
(8)	$\underline{(s_a, \sigma_4^{dy})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (i_c, \sigma_6) \Rightarrow$	$(PS_1^{\rightarrow +})$
(9)	$\underline{(i_a, \sigma_7)}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel \underline{(i_c, \sigma_6)} \Rightarrow$	(IF_{true})
(10)	$(i_a, \sigma_7).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel \underline{(q_{c1}, \sigma_6)} \Rightarrow$	(ZU)
(11)	$(i_a, \sigma_7).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel \underline{(q_{c2}, \sigma_8)} \Rightarrow$	(IF_{true_r})
(12)	$(i_a, \sigma_7).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel \underline{(re_c, \sigma_8)} \Rightarrow$	$(APP_r^{\rightarrow +})$
(13)	$\underline{(i_a, \sigma_7)}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	(ZU)
(14)	$\underline{(q_{a1}, \sigma_9)}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	(IF_{false})
(15)	$\underline{(q_{a4}, \sigma_9)}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$(PA_1^{\rightarrow +})$
(16)	$(q_{a5}, \sigma_{10}) \parallel \underline{(s_c, \sigma_6^{dy})}).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$(APP_i^{\rightarrow +})$
(17)	$(q_{a5}, \sigma_{10}) \parallel \underline{(i_c, \sigma_{12})}).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	(IF_{true})
(18)	$(q_{a5}, \sigma_{10}) \parallel \underline{(q_{c1}, \sigma_{12})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	(ZU)
(19)	$(q_{a5}, \sigma_{10}) \parallel \underline{(q_{c2}, \sigma_{13})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	(IF_{true_r})
(20)	$(q_{a5}, \sigma_{10}) \parallel \underline{(re_c, \sigma_{13})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$(APP_r^{\rightarrow +})$
(21)	$\underline{(q_{a5}, \sigma_{10}) \parallel (r_c, \sigma_7^{dy})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$(PAR_{sync}^{\rightarrow +})$
(22)	$\underline{(q_{a6}, \sigma_{11})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	(IF_{false_r})
(23)	$\underline{(q_{a7}, \sigma_{11})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	(IF_{true})
(24)	$(q_{a8}, \sigma_{11}).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy})$	$Normalform$

Abbildung 4.32.: Lauf der operationalen Semantik auf das Service-orientierte System in Abbildung 4.26 bei $x := 2$

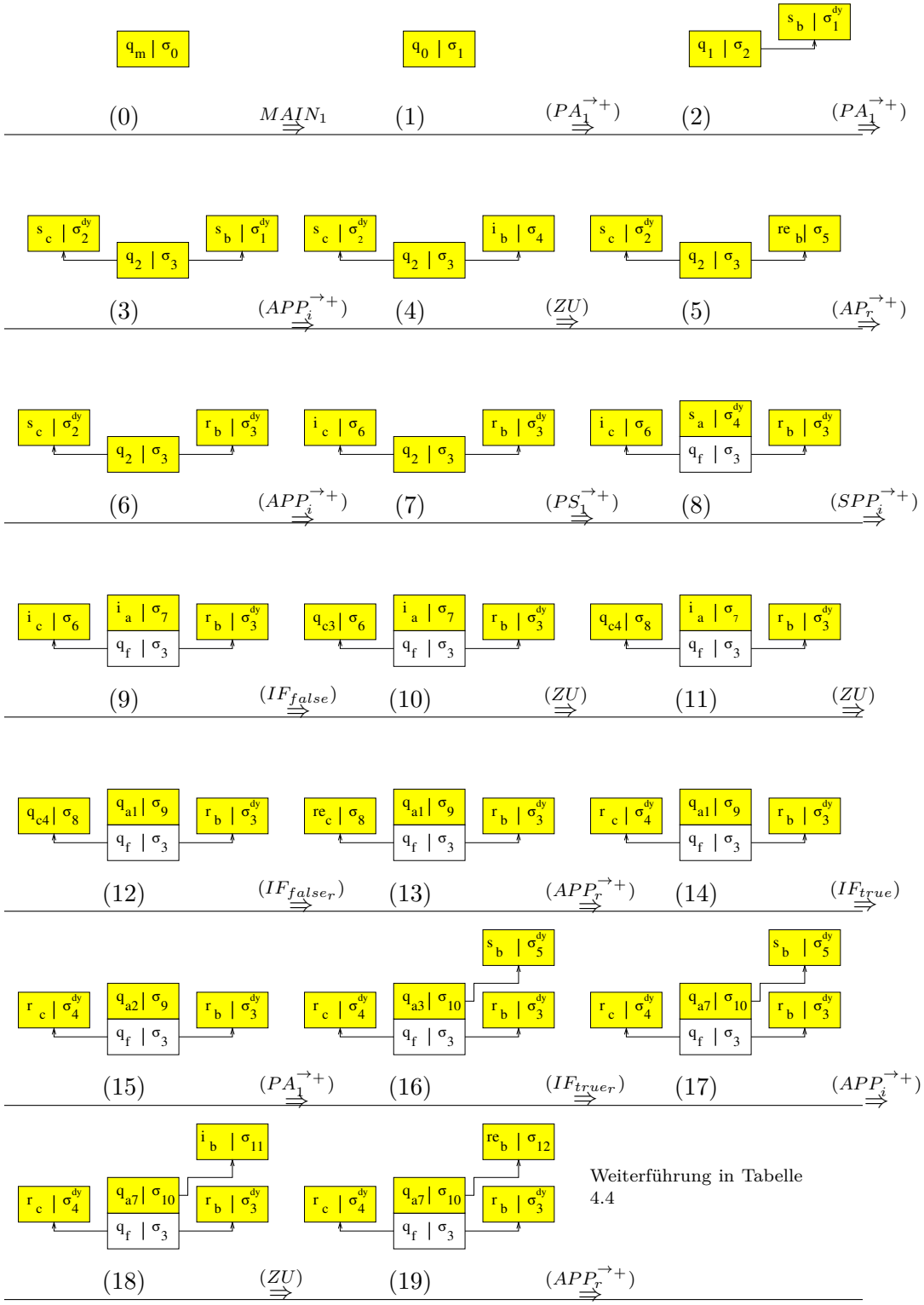


Tabelle 4.3.: Anwendung der Regeln der operationalen Kaktuskeller-Semantik auf das Service-orientierte System in Abbildung 4.26.

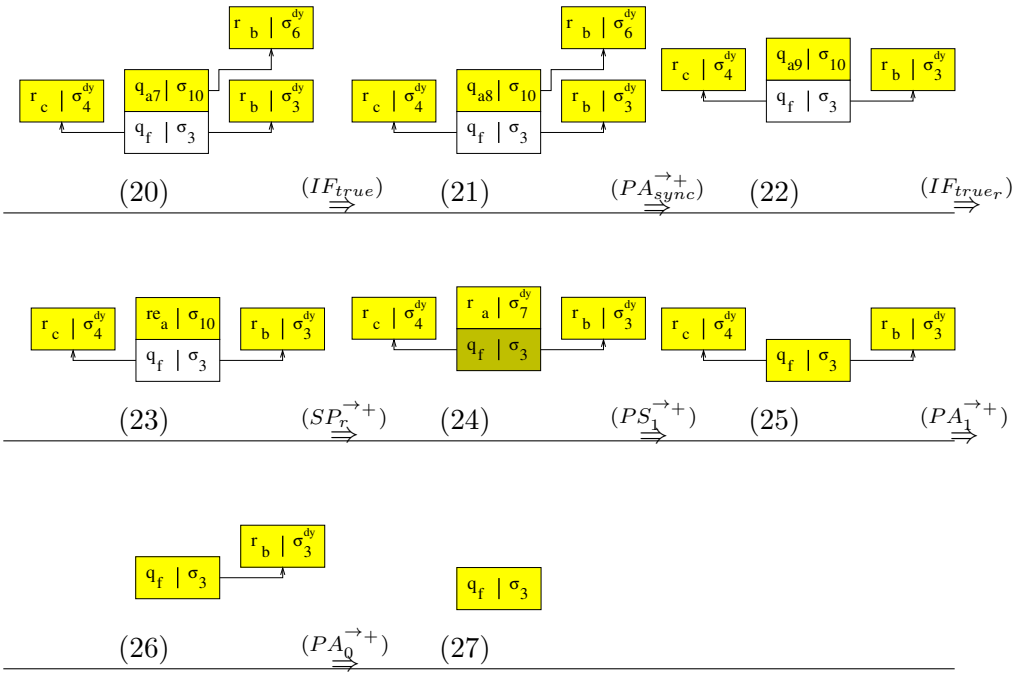


Tabelle 4.4.: Teil 2: Grafische Darstellung der Anwendung der Regeln der operationalen Kaktuskeller-Semantik auf das Service-orientierte System *SoS* in Abbildung 4.26 bei Aufruf mit $x := 1$. Abbau der Kaktuskeller.

Speicher	Inhalt	Dummy-Speicher	Inhalt
σ_0	[]	σ_1^{dy}	$[b := 1, x_0 := 2]$
σ_1	$[x := 2]$	σ_2^{dy}	$[c := 2, x_0 := 2]$
σ_2	$[x := 2, f := (b, 1)]$	σ_3^{dy}	$[b := 1]$
σ_3	$[x := 2, f := (b, 1), g := (c, 2)]$	σ_4^{dy}	$[x_0 := 2]$
σ_4	$[b := 1, y := 2]$	σ_5^{dy}	$[c := 2, rval := 1]$
σ_5	$[b := 1, y := 3]$	σ_6^{dy}	$[c := 3, x_0 := 2]$
σ_6	$[c := 2, v := 2]$	σ_7^{dy}	$[c := 3, rval := 1]$
σ_7	$[z := 2]$		
σ_8	$[c := 2, v := 1]$		
σ_9	$[z := 2, x := 2]$		
σ_{10}	$[z := 2, x := 2, g := (c, 3)]$		
σ_{11}	$[z := 2, x := 1, g := (c, 3)]$		
σ_{12}	$[c := 3, v := 2]$		
σ_{13}	$[c := 3, v := 1]$		

Abbildung 4.33.: Der Inhalt der Speicher und Dummy-Speicher aus Abbildung 4.32.

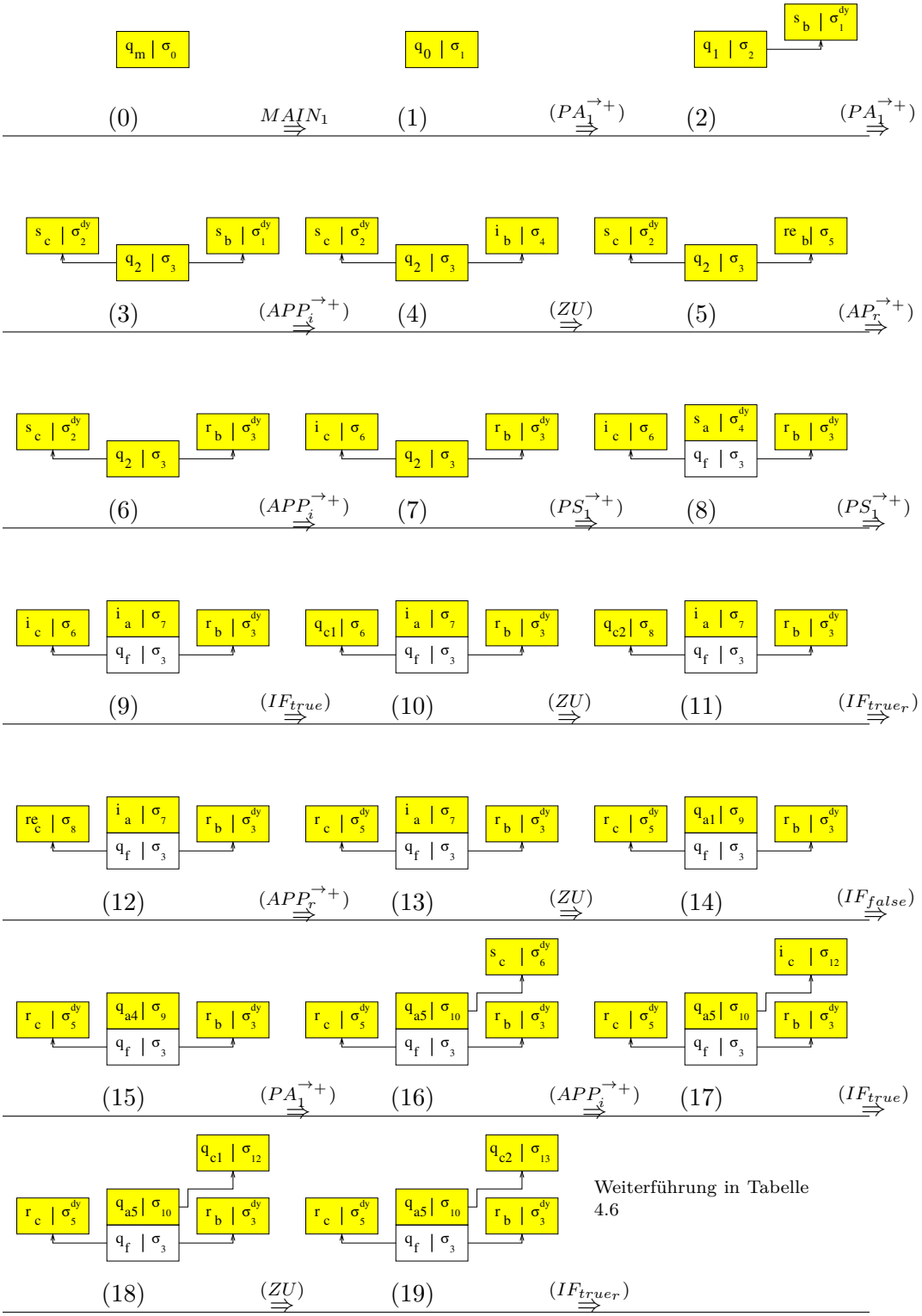


Tabelle 4.5.: Grafische Darstellung der Anwendung der Regeln der operationalen Kaktuskeller-Semantik auf das Service-orientierte System *SoS* in Abbildung 4.26 bei Aufruf mit $x := 2$.

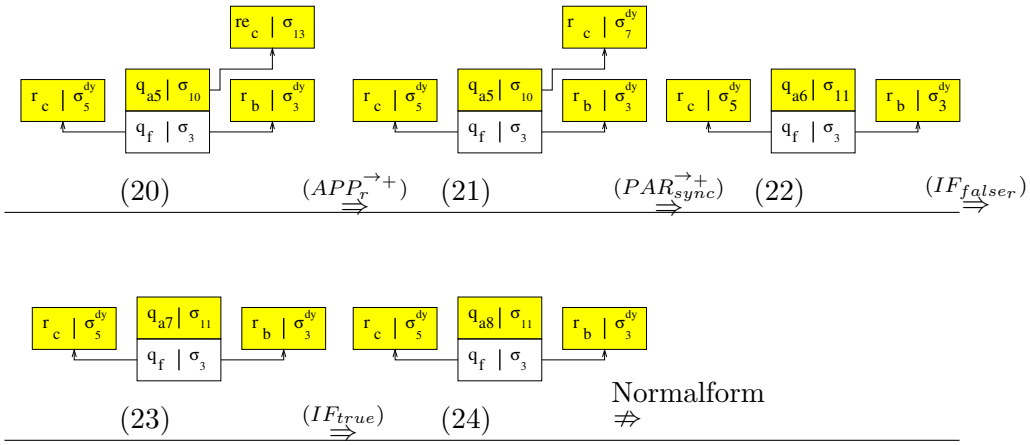


Tabelle 4.6.: Teil 2: Grafische Darstellung der Anwendung der Regeln der operationalen Kaktuskeller-Semantik auf das Service-orientierte System *SoS* in Abbildung 4.26 bei Aufruf mit $x := 2$. Abbau der Kaktuskeller.

5. Randbedingungen und Grenzen Petri-Netz-basierter Ansätze

Eine exakte statische Analyse auf Basis der formalen Semantik ist algorithmisch nicht berechenbar. Aus diesem Grund existierten eine Reihe von Programmanalyseverfahren [11] (abstraktions-basiert), [46] (verfeinerungs-basiert), um Aussagen über gewisse Eigenschaften, wie zum Beispiel die Deadlockfreiheit [22], [2], von Programmen treffen zu können. In diesem Kapitel wird untersucht, inwieweit Petri-Netze als formales Modell für die statische Analyse von Programmen auf Deadlockfreiheit genutzt werden können.

5.1. Abstraktion

Die Idee zur abstraktions-basierten Analyse liegt darin, die konkrete Semantik eines Programms in ein berechenbares Modell zu abstrahieren, dabei entspricht jede Ausführung in der konkreten Semantik einer Ausführung in der abstrakten Semantik, siehe Abbildung 5.1. Zur Darstellung des Verhalten von konkreten Systemen werden als formales Modell die Petri-Netze verwendet [45]. Auf Petri-Netzen ist die Deadlockanalyse, als statische Analyse, entscheidbar und damit, im Gegensatz zum konkreten System, berechenbar.

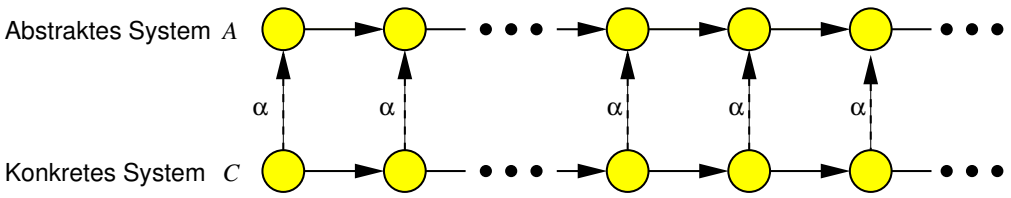


Abbildung 5.1.: Abstraktion

Die Abstraktion eines konkreten Systems in ein abstraktes System, wird durch die Anwendung einer Funktion α realisiert, vgl. mit Abbildung 5.1. Dabei soll jeder unerwünschte Zustand (zum Beispiel ein Deadlock) in der konkreten Semantik bzw. System auch in einen unerwünschten Zustand in der abstrakten Semantik bzw. im abstrakten System führen.

Zur Erläuterung der Abstraktion eines konkreten Systems in ein abstraktes System wird die Definition eines Zustandsübergangssystems benötigt.

Definition 5.1 (Zustandsübergangssystem)**.** Ein Zustandsübergangssystem (engl. transition system) ist ein Tupel $Z \widehat{=} (Q, \rightarrow, I, F)$, wobei

- Q eine Menge von Zuständen ist,
- $I \subseteq Q$ eine Menge von Anfangszuständen ist,
- $F \subseteq Q$ eine Menge von Finalzuständen ist und
- $\rightarrow \subseteq Q \times Q$ eine Zustandsübergangsrelation ist, so dass für alle $s \in F$ gilt: Es gibt kein $s' \in Q$ mit $s \rightarrow s'$.

Ein Lauf von Z (engl. run) ist eine endliche oder unendliche Folge $\langle s_i : 0 \leq i < k \rangle$, $k \in \mathbb{N} \cup \{\infty\}$, so dass $s_0 \in I$ und $s_{i-1} \rightarrow s_i$ für $1 \leq i < k$. Ein Zustand $s \in Q$ heißt unerwünscht gdw. es kein $s' \in Q$ mit $s \rightarrow s'$ gibt und $s \notin F$ ist.

Notation: $\stackrel{*}{\Rightarrow}$ ist die reflexive, transitive Hülle von $\rightarrow$.

Das konkrete Verhalten eines konkreten Systems kann als Zustandsübergangssystem dargestellt werden, vgl. Kapitel 4.

Basierend auf zwei Zustandsübergangssystemen kann nun die Abstraktion definiert werden:

Definition 5.2 (Abstraktion)**.** Seien $Z_1 \widehat{=} (Q_1, \rightarrow_1, I_1, F_1)$ und $Z_2 \widehat{=} (Q_2, \rightarrow_2, I_2, F_2)$ zwei Zustandsübergangssysteme. Z_2 ist die Abstraktion von Z_1 genau dann wenn es eine Funktion $\alpha : Q_1 \rightarrow Q_2$ (Abstraktionsfunktion) mit folgenden Eigenschaften gibt:

i Für alle $q \in I_1$ gilt: $\alpha(q) \in I_2$

ii Für alle $q \in F_1$ gilt: $\alpha(q) \in F_2$.

iii Falls $q \rightarrow_1 q'$, dann gilt auch $\alpha(q) \rightarrow_2 \alpha(q')$

Theorem 5.1 (Läufe in Abstraktion)**.** *Sei ein Zustandsübergangssystem $Z_2 \widehat{=} (Q_2, \rightarrow_2, I_2, F_2)$ eine Abstraktion vom Zustandsübergangssystem $Z_1 \widehat{=} (Q_1, \rightarrow_1, I_1, F_1)$ mit Abstraktionsfunktion $\alpha : Q_1 \rightarrow Q_2$. Für jeden Lauf $\langle q_i : 0 \leq i < k \rangle$, $k \in \mathbb{N} \cup \{\infty\}$ von Z_1 ist $\langle \alpha(q_i) : 0 \leq i < k \rangle$, $k \in \mathbb{N} \cup \{\infty\}$ ein Lauf von Z_2.*

Beweis 5.1. Jeder Lauf in Z_1 beginnt mit einem Anfangszustand q_0. Wegen der Eigenschaft i in der Definition 5.2 gilt, dass $\alpha(q_0) \in I_2$ ist.

Wegen Punkt (iii) in der Definition 5.2 zur Abstraktion gilt, dass $\alpha(q_{i-1}) \rightarrow_2 \alpha(q_i)$ für alle $0 \leq i < k$. Daraus folgt, dass auch $\langle \alpha(q_i) : 0 \leq i < k \rangle$ ein Lauf in Z_2 sein muss.

Beispiel 5.1. Für das in Beispiel 4.2 definierte Zustandsübergangssystem ist

$SAB = (Z, \Rightarrow_{SAB}, q, F)$, dabei sei:

- $Z = \{(q_m, \sigma_0), (q_0, \sigma_1), (q_1, \sigma_2), (i_a, \sigma_3).(q_f, \sigma_2), (q_{a1}, \sigma_4).(q_f, \sigma_2), (re_a, \sigma_5) \parallel (i_b, \sigma_6).(q_f, \sigma_2), (re_a, \sigma_5) \parallel (re_b, \sigma_7).(q_f, \sigma_2), (re_a, \sigma_5).(q_f, \sigma_2), (q_f, \sigma_8)\}$,
- $\Rightarrow_{SAB}$ dargestellt in Abbildung 5.2,
- $q \hat{=} (q_m, \sigma_0)$,
- $F \hat{=} \{(q_f, \sigma_8)\}$.

Für das Zustandsübergangssystem SAB wird eine Abstraktionsfunktion eingeführt: $\alpha : PEX(SFRAME) \rightarrow PEX(NODE)$. Die Abstraktionsfunktion ist weiterhin definiert durch $\alpha(q_f) = f \in F_a$, $\alpha(e, \sigma) \hat{=} \alpha'(e)$, wobei e ein prozess-algebraischer Ausdruck über Q_1 ist. α' sei definiert als:

- $\alpha'(e) = q$ mit $q \in NODE$,
- $\alpha'(e_1.e_2) \hat{=} \alpha'(e_1).\alpha'(e_2)$ und
- $\alpha'(e_1||e_2) \hat{=} \alpha'(e_1)||\alpha'(e_2)$.

e_1, e_2 und e sind prozess-algebraische Ausdrücke über $PEX(NODE)$.

Durch Anwendung der Abstraktionsfunktion wird die abstrakte Semantik in Form des Zustandsübergangssystems $[\![SAB]\!]_{abstrakt} \hat{=} (Z_2, \Rightarrow_{SAB_2}, q_2, F_2)$ erhalten, wobei:

- $Z_2 = \{q_m, q_0, q_1, i_a.q_f, q_{a1}.q_f, re_a \parallel i_b.q_f, re_a \parallel re_b.q_f, re_a.q_f, q_f\}$,
- $\Rightarrow_{SAB_2}$ dargestellt in Abbildung 5.2,
- $q \hat{=} q_m$,
- $F_2 \hat{=} \{q_f\}$.

Regeln im konkreten System $[\![SAB]\!]_{konkret}$	Regeln im abstrakten System $[\![SAB]\!]_{abstrakt}$
$\frac{}{(q_m,\sigma_0) \to (q_0, \sigma_1\vert_x^{eval^{\sigma_0}(x)})}(MAIN_1)$	$\frac{}{q_m \to q_0}(MAIN_1)$
$\frac{}{(q_0,\sigma_1) \to (q_1, \sigma_2\vert_x^{eval(3)})}(ZU)$	$\frac{}{q_0 \to q_1}(ZU)$
$\frac{}{(q_1,\sigma_2) \to (i_a, \sigma_3\vert_y^{eval^{\sigma_2}(x)}).(q_f,\sigma_2)}(PSR_1)$	$\frac{}{q_1 \to i_a.q_f}(PSR_1)$
$\frac{}{(i_a,\sigma_3) \to (q_{a1}, \sigma_4\vert_y^{eval(1)})}(ZU)$	$\frac{}{i_a \to q_{a1}}(ZU)$
$\frac{}{(q_{a1},\sigma_4) \to (re_a, \sigma_5\vert_f^{(b,1)}) \vert\vert (i_b, \sigma_6\vert_{z,b}^{eval^{\sigma_4}(y),1})}(PA_1)$	$\frac{}{q_{a1} \to re_a \vert\vert i_b}(PA_1)$
$\frac{}{(i_b,\sigma_6) \to (re_b, \sigma_7\vert_z^{eval(2)})}(ZU)$	$\frac{}{i_b \to re_b}(ZU)$
$\frac{\sigma_5(f) = (b, \sigma_7(b))}{(re_a,\sigma_5) \vert\vert (re_b,\sigma_7) \to (re_a,\sigma_5)}(PA_1)$	$\frac{}{re_a \vert\vert re_b \to re_a}(PA_1)$
$\frac{}{(re_a,\sigma_5).(q_f,\sigma_2) \to (q_f, \sigma_8\vert_y^{eval^{\sigma_5}(y)})}(PSR_1)$	$\frac{}{re_a.q_f \to q_f}(PSR_1)$

Abbildung 5.2.: Regeln der konkreten Semantik (operationalen Semantik) und der abstrakten Semantik des Service-orientierten Systems in Abbildung 4.11

In Abbildung 5.3 wird der Lauf bei gleicher Eingabe zum einen im abstrakten System und zum anderen im konkreten Zustandsübergangssystem gezeigt. Es wird deutlich, dass durch die Anwendung der Abstraktionsfunktion auf die Regeln des Zustandsübergangssystems $[\![SAB]\!]_{konkret}$ der Speicher verloren geht. In der Abbildung 5.2 wird deutlich, dass nicht nur der Speicher verloren geht, sondern auch die Voraussetzungen der Regeln, was zur Folge hat, dass bei der gleichen Eingabe mehr Läufe möglich sind. Diese möglichen abstrakten Läufe haben keine entsprechenden konkreten Lauf.

Lauf im konkreten System $[\![SAB]\!]_{konkret}$	Lauf im abstrakten System $[\![SAB]\!]_{abstrakt}$
$\underline{(q_m,\sigma_0)} \Rightarrow$	$\underline{q_m} \Rightarrow$
$\underline{(q_0,\sigma_1)} \Rightarrow$	$\underline{q_0} \Rightarrow$
$\underline{(q_1,\sigma_2)} \Rightarrow$	$\underline{q_1} \Rightarrow$
$\underline{(i_a,\sigma_3)}.(q_f,\sigma_2) \Rightarrow$	$\underline{i_a}.q_f \Rightarrow$
$\underline{(q_{a1},\sigma_4)}.(q_f,\sigma_2) \Rightarrow$	$\underline{q_{a1}}.q_f \Rightarrow$
$(re_a,\sigma_5) \parallel \underline{(i_b,\sigma_6)}.(q_f,\sigma_2) \Rightarrow$	$re_a \parallel \underline{i_b}.q_f \Rightarrow$
$\underline{(re_a,\sigma_5) \parallel (re_b,\sigma_7)}.(q_f,\sigma_2) \Rightarrow$	$\underline{re_a \parallel re_b}.q_f \Rightarrow$
$\underline{(re_a,\sigma_5).(q_f,\sigma_2)} \Rightarrow$	$\underline{re_a.q_f} \Rightarrow)$
(q_f,σ_8)	q_f

Abbildung 5.3.: Lauf in der konkreten (operationalen) und abstrakten Semantik des Service-orientierten Systems in Abbildung 4.11.

Da die Inferenzregeln der abstrakten Semantik keine dynamische Information wie den Speicher enthalten, können diese Inferenzregeln auch direkt aus der Implementierung des Programms π bestimmt werden.

Die in diesem Abschnitt vorgestellte Abstraktion kann soweit angepasst werden, dass die abstrakte Semantik auf einem bestimmten formalen Modell basiert. Wie bereits erwähnt wurde, existieren eine Reihe von formalen Modellen. Mögliche Modelle sind endliche

Automaten, die sequentielles Verhalten darstellen können, oder Keller, die rekursiver Verhalten darstellen können. Parallelität und Nebenläufigkeit können nicht mit diesen Modellen dargestellt werden und werden daher in diesem Kapitel nicht weiter betrachtet. Petri-Netze können Parallelität und Nebenläufigkeit darstellen. Aus diesem Grund wird im folgenden Abschnitt die Abstraktionsfunktion so angepasst, dass das abstrakte System auf Petri-Netzen basiert.

5.2. Petri-Netz-Abstraktion

Die Abstraktion eines konkreten Programms zu (P,P)-PRS-Regeln wird bereits im Grundlagenkapitel unter 3.3.1 Abstraktionsverfahren erläutert. In [38], Kapitel 2, zeigt Mayr, dass (P,P)-PRS äquivalent zu Petri-Netzen sind.

Die Abstraktion basiert darauf, dass die Programmpunkte den Stellen entsprechen. Jede Prozedur p hat einen eindeutigen Eintritts- und Austrittspunkt. Die Eintrittspunkte bzw. die Eintrittsstellen werden mit i_p und die Austrittspunkte bzw. Austrittsstellen mit r_p gekennzeichnet. Für externe Prozedur- oder Funktionsaufrufe werden diese Stellen mit dem Namen des Platzhalter $init_p$ und ret_p gekennzeichnet. Bei der Komposition werden diese Platzhalter über eine Platzhalterfunktion, ähnlich wie bei der operationalen Semantik, mit dem Eintritts- und Austrittspunkt oder Stelle der aufgerufenen externen Funktion identifiziert.

Die Anfangsmarkierung kennzeichnet den Zustand vor Ausführung des Programms. Es befindet sich eine Marke in der Stelle q_0, die den Eintrittspunkt der *main*-Funktion repräsentiert. Werden asynchrone Funktionen oder Prozeduren aufgerufen, so wird dies als Verzweigung in der entsprechenden Petri-Netz Abstraktion dargestellt. Beim Aufruf einer asynchronen Prozedur oder Funktion wird damit aus einer Marke zwei Marken. Falls Aufrufer und aufgerufene asynchrone Prozeduren beide am jeweiligen *return* angekommen sind, wird synchronisiert, vgl. Abbildung 5.1. Erreicht der Aufrufer eine *sync*-Anweisung, muss dieser warten, bis die asynchron aufgerufenen Prozedur die *return*-Anweisung erreicht.

Die Kompositionalität wird analog zu Workflow Netzen, eine Arbeit von van der Aalst et al.[2], durchgeführt.

Die Regeln der Petri-Netz-Abstraktionen werden durch die Anwendung der folgenden Abstraktionsfunktion erhalten:

Definition 5.3 (Petri-Netz-Abstraktionsfunktion)**.** Die Abstraktionsfunktion sei $\alpha_{pn} : PEX(SFRAME) \rightarrow PEX(NODE)$ und sei weiterhin definiert durch $\alpha_{pn}(q_f) = f \in F$, $\alpha_{pn}(e, \sigma) \hat{=} \alpha'_{pn}(e)$, wobei e ein prozess-algebraischer Ausdruck über Q ist. α'_{pn} sei definiert

als:

- $\alpha'_{pn}(e) = q$ mit $q \in NODE$,
- $\alpha'_{pn}(e_1.e_2) \widehat{=} \alpha'(e_1)$ und
- $\alpha'_{pn}(e_1||e_2) \widehat{=} \alpha'_{pn}(e_1)||\alpha'_{pn}(e_2)$.

e_1, e_2 und e sind prozess-algebraische Ausdrücke über $PEX(NODE)$ und $q \in NODE$.

Die Anwendung der Abstraktionsfunktion α_{pn} auf die Regeln der operationalen Semantik wird in Tabelle 5.1 gezeigt. In Spalte 1 werden die wichtigsten Kontrollstrukturen gezeigt. Spalte 2 zeigt die Inferenzregel der konkreten Semantik, die das Verhalten der Kontrollstruktur zeigt. In der dritten Spalte wird die Regel der abstrakten Semantik gezeigt, die sich aus der Anwendung von α_{pn} ergibt. Die vierte Spalte zeigt die aus Spalte 3 resultierende Petri-Netz-Darstellung.

Die Abstraktionen für Services werden ebenfalls so erstellt. Für service-übergreifende Prozedur- und Funktionsaufrufe werden Platzhalter in der Abstraktion eingeführt, die im nächsten Kapitel beschrieben werden.

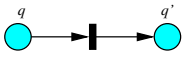
5.2.1. Abstraktion von Services

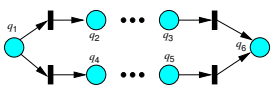
Zur Darstellung des abstrakten Verhaltens von Services wird die Tabelle zur Abstraktion der Kontrollstrukturen in monolithischen Programmen (Tab. 5.1) erweitert, um die Abstraktion von grenzübergreifenden Prozedur- und Funktionsaufrufen darstellen zu können.

Dabei werden, wie eingangs erwähnt, für externe Prozedur- oder Funktionsaufrufe Stellen mit dem Namen des Platzhalter $init_p$ und ret_p eingeführt. Diese Platzhalter werden bei der Komposition der Services über die Platzhalterfunktion $\mathcal{D}$, die die Services mit Angebotsschnittstellen zur Verfügung stellen, aufgelöst. Die Platzhalterfunktion $\mathcal{D}$ in Tabelle 5.2 werden definiert über $\mathcal{D}(init_p) = i_p$ und $\mathcal{D}(ret_p) = r_p$.

Die grafische Darstellung der Petri-Netze ist analog zu den Petri-Netzen der internen Prozedur- und Funktionsaufrufen, lediglich die Benennung der Stellen ist unterschiedlich. Aus der Stelle i_p wird die Stelle $init_p$ und aus den Stellen r_p wird ret_p.

Bemerkung 5.1. *Bei der Darstellung des konkreten Verhaltens von Service-orientierten Systemen wurden in Kapitel 4 die zusätzlichen Programmpunkte in Definition 4.18 eingeführt. Diese Programmpunkte sind im konkreten Verhalten ein Hilfsmittel, zur Übertra-*

Kontrollstruktur	Inferenzregel	Anwendung von α_{pn}	Darstellung als Petri-Netz [38], Kap.2
q : $x := e;$ q' : ...	$\frac{}{(q,\sigma) \to (q', \sigma\vert_x^{eval_A^\sigma(e)})}$	$\frac{}{q \to q'}$	

q_1 : **if** e{
q_2 : ...
q_3 : ...}
else{
q_4 : ...
q_5 : ...}
q_6 : ...
	$\frac{eval_B^\sigma(e) = true}{(q_1,\sigma) \to (q_2,\sigma)}$
	$\frac{eval_B^\sigma(e) = false}{(q_1,\sigma) \to (q_4,\sigma)}$
	$\frac{}{(q_3,\sigma) \to (q_6,\sigma)}$
	$\frac{}{(q_5,\sigma) \to (q_6,\sigma)}$
		$\frac{}{q_1 \to q_2}$
		$\frac{}{q_1 \to q_4}$
		$\frac{}{q_3 \to q_6}$
		$\frac{}{q_5 \to q_6}$
			

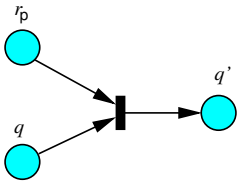
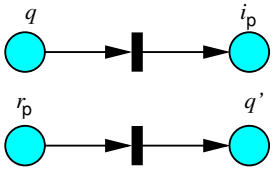
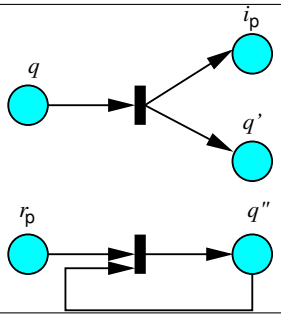
Synchronisation q : **sync** p; q' : ... p{ i_p : ... r_p : **return;**}	$\frac{\sigma(f) = (q, \sigma'(p))}{(r_p,\sigma')\vert\vert(q,\sigma) \to (q',\sigma)}$	$\frac{}{r_p \Vert q \to q'}$	
Synchrone Prozedur p q : p; q' : ... $p()${ i_p : ... r_p : **return;**}	$\frac{}{(q,\sigma) \to (i_p,\sigma').(q',\sigma)}$, $\frac{}{(r_p,\sigma').(q',\sigma) \to (q',\sigma)}$	$\frac{}{q \to i_p}$ $\frac{}{r_p \to q'}$	
asynchrone Prozedur p a{... q : p; q' : ... q'' : **return;**} $p()${ i_p : ... r_p : **return;**}	$\frac{}{(q,\sigma) \to (q', \sigma\vert_f^{(p,\Phi)})\vert\vert\vert(i_p, \sigma''\vert_p^\Phi)}$, $\frac{\sigma(f) = (p, \sigma'(p))}{(q'',\sigma)\vert\vert(r_p,\sigma') \to (q'',\sigma)}$	$\frac{}{q \to q' \Vert i_p}$ $\frac{}{q'' \Vert r_p \to q''}$	

Tabelle 5.1.: Abstraktion von Kontrollstrukturen zu Petri-Netzen [54].

Kontrollstruktur	Inferenzregel	Anwendung von α_{pn}
Synchrone Prozedur p q : p; q' : $\cdots$ $p()\{$ i_p : $\cdots$ r_p : **return;**$\}$	$\frac{}{(q,\sigma) \to (init_p^{R_A}, \sigma_i^{dy}).(q',\sigma)}$, $\frac{}{(ret_p^{R_A}, \sigma_r^{dy}).(q',\sigma) \to (q', \sigma\vert_y^{eval^{\sigma_r^{dy}}(rval)})}$	$\frac{}{q \to init_p}$ $\frac{}{ret_p \to q'}$
asynchrone Prozedur p $a\{\cdots$ q : p; q' : $\cdots$ q'' : **return;**$\}$ $p()\{$ i_p : $\cdots$ r_p : **return;**$\}$	$\frac{}{(q,\sigma) \to (q', \sigma'\vert_f^{(p,n)}) \vert\vert (init_p^{R_A}, \sigma_i^{dy}\vert_p^n)}$, $\frac{\sigma(f) = (p, \sigma_r^{dy}(p))}{(q'',\sigma) \parallel (ret_p^{R_A}, \sigma_r^{dy}) \to (q',\sigma)}$	$\frac{}{q \to q' \parallel init_p}$ $\frac{}{q'' \parallel ret_p \to q''}$
Synchronisation q : **sync** p; q' : $\cdots$ $p\{$ i_p : $\cdots$ r_p : **return;**$\}$	$\frac{\sigma(f) = (p, \sigma_r^{dy}(p))}{(q,\sigma) \parallel (ret_p^{R_A}, \sigma_r^{dy}) \to (q',\sigma)}$	$\frac{}{q \parallel ret_p \to q'}$

Tabelle 5.2.: Abstraktion von service-übergreifenden Prozeduraufrufen. Platzhalterfunktion $\mathcal{D}$ ist definiert durch $\mathcal{D}(init_p) = i_p$ und $\mathcal{D}(ret_p) = r_p$.

gung von Parametern. Da die Regeln zu Überführung des Speichers von der Form $a \to a'$ wobei a und a' atomare Prozesse sind und die Transitivität gilt, geht dieser Schritt verloren und die Platzhalterfunktion bildet auf den Ein- und Austrittspunkt der Prozeduren.

Daraus ergibt sich die Definition für das abstrakte Verhalten basierend auf Petri-Netzen:

Definition 5.4 (Abstraktes Verhalten mit Petri-Netz-basierten Abstraktionen)**.** Sei S eine Komponente mit Angebotsschnittstellen $I_1, \ldots, I_k$ und Nutzungsschnittstellen $R_1, \cdots, R_m$. Weiterhin besitzt S eine Implementierung π mit Programmpunkten $NODE_\pi$. Eine Petri-Netz-basierte Abstraktion von S ist ein Tupel $PN(S) \widehat{=} (NNODE, \mathcal{D}, [\![\pi]\!], \mu_0, \mu_f)$, wobei

- $DNODE \widehat{=} DNODE_I \cup DNODE_R$ und $DNODE_I \widehat{=}$
- $\mathcal{D} : DNODE_I \to NODE_\pi$, $\mathcal{D}(init_p) = i_p$, wobei i_p der Eintrittspunkt von p ist und $\mathcal{D}(ret_p) = r_p$, wobei r_p Austrittspunkt von p ist,

- $[\![\pi]\!]$ ist (sind) die Netz(e), die aus den Regeln in Tab. 5.1 und 5.2 aus π entstanden ist (sind),
- $\mu_0 : NODE_\pi \to \mathbb{N}$ die Anfangsmarkierung ist, so dass $\mu_0(i_{main}) = 1$ und $\mu_0(q) = 0$ für $q \neq i_{main}$
- $\mu_f : NODE_\pi \to \mathbb{N}$ die Finalmarkierung ist, so dass $\mu_f(r_{main}) = 1$ und $\mu_f(q) = 0$ für $q \neq r_{main}$

5.2.2. Komposition der Service-Abstraktionen

Wenn Abstraktionen zur Darstellung des abstrakten Verhaltens aller Services im Service-orientierten System vorhanden sind, können die einzelnen Petri-Netz-basierten Abstraktionen nach der Definition 5.5 miteinander verknüpft werden.

Definition 5.5 (Komposition von Petri-Netz-basierter Abstraktionen)**.** Seien $S_1, \cdots, S_n$ die Services mit Implementierungen π, Angebotsschnittstellen I_i und Nutzungsschnittstellen R_i, dann ist $SoS[S_1, \ldots, S_n]$ ein Service-orientiertes System sowie $PN(S_i) \widehat{=} (NNODE^{(i)}, \mathcal{D}_i, [\![\pi_i]\!], \mu_0^{(i)}, \mu_f^{(i)})$ mit $i = 1, \cdots, n$ und $n \in \mathbb{N}$. Die Komposition der Petri-Netz-basierten Abstraktionen ist ein Petri-Netz $SoS[P(S_1, \ldots, S_n)] \widehat{=} (S, T, \mu_0)$, wobei $(S, T) \widehat{=} \mathcal{D}([\![\pi_1]\!]) \cup \cdots \cup \mathcal{D}([\![\pi_n]\!])$ und

$$\mathcal{D}(q) \widehat{=} \begin{cases} \mathcal{D}_j(init_p), & \text{falls } q = init_p, p \in R_i \cap I_j \text{ und } N_i \to I_j \in SoS \\ \mathcal{D}_j(ret_p), & \text{falls } q = ret_p, p \in R_i \cap I_j \text{ und } R_i \to I_j \in SoS \\ q, & \text{sonst} \end{cases} .$$

und $\mu_0 \widehat{=} \mu_0^{(1)} \cup \cdots \cup \mu_0^{(n)}$.

Durch die Komposition der Services entsteht ein System, dass das abstrakte Verhalten des zugrundeliegenden Service-orientierten Systems beschreibt.

5.2.3. Zusammenfassung

Im zurückliegenden Abschnitt wurde die Abstraktionsfunktion, zu Erstellung des abstrakten Verhaltens aus dem konkreten Verhalten eines Programms bzw. für Services gezeigt. Dabei wurden die Zustandsübergangssysteme um Platzhalter erweitert zu Petri-Netz-Abstraktionen von Services. Weiterhin wurde definiert, wie das abstrakte Verhalten von Services vereinigt werden kann, zur Komposition der Services zu einem System, welches das abstrakte Verhalten eines Service-orientierten Systems beschreibt. Im nächsten Schritt wird anhand eines Beispiels gezeigt, wie die Abstraktion und Komposition durchgeführt. Auf dem gegebenen Beispiel wird eine Deadlockanalyse, das Erreichen

eines unerwünschten Zustandes, ausgeführt, um die Grenzen PN-basierter Ansätze zu zeigen.

5.3. Deadlockanalyse

Theorem 5.2 (Erhaltung der Deadlocksituation in Petri-Netz-Abstraktionen)**.** *Wenn e in Normalform vorliegt, dann liegt auch $\alpha(e)$ in Normalform vor, wobei die Funktion α_{pn} die (P,P)-Abstraktion darstellt.*

Beispiel 5.2. Dazu wird das Service-orientierte System aus Abbildung 4.26 aus Kapitel 4 erneut betrachtet, um die (P,P)-PRS-basierte Abstraktionen der Services M, A, B und C zu erstellen.

(P,P)-PRS-basierte Abstraktion der Services

Durch Anwendung der Regeln aus Tabelle 5.1 und 5.2 werden folgenden Petri-Netze für das Beispiel aus Abbildung 4.26 erhalten.

Die Abstraktion von Service M ist in Abbildung 5.4 zu sehen. Die Programmpunkte q_0, q_1, q_2 und q_f werden von den Stellen q_0, q_1, q_2 und q_f repräsentiert. Es ergeben sich zwei Petri-Netze. Grund dafür ist der synchrone Aufruf der Prozedur a. Für den Aufruf von a wird die Stelle $init_a$. Für die Rückkehr von a wird die Stelle ret_a eingeführt. Die Aufrufe der asynchronen Funktionen b und c werden durch die Stellen $init_b$ und $init_c$ repräsentiert. Die Parallelität wird durch die Verzweigung dargestellt.

Das Petri-Netz welches den Service A repräsentiert wird in Abbildung 5.5 gezeigt. Es beinhaltet für jeden Programmpunkt im Service A eine Stelle und für die asynchronen Aufrufe von b und c die Stellen $init_b$ und $init_c$ sowie die Stellen ret_b und ret_c.

Da der Service B nur aus den Ein- und Austrittsprogrammpunkten besteht, besitzt das resultierende Petri-Netz aus der Stelle i_b und r_b sowie einer Transition, die beide Stellen miteinander verbindet und die Zuweisung am Programmpunkt i_a repräsentiert. Das Petri-Netz zu Service B ist in Abbildung 5.6 zu sehen.

In Abbildung 5.7 wird das Petri-Netz von Service C gezeigt. Es hat, wie das Petri-Netz von Service B, eine Stelle $init_c$ für den Aufruf der zur Verfügung gestellten Funktion c und eine Stelle ret_c für die Rückkehr von c. Für jeden Programmpunkt in Service C existiert eine Stelle in dem Petri-Netz.

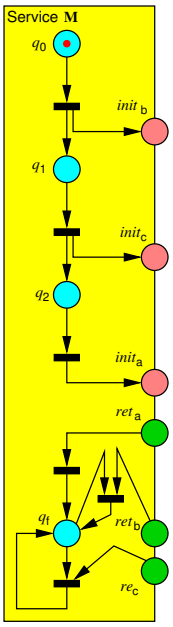


Abbildung 5.4.: Die Petri-Netz-basierte Abstraktion von Service M aus dem Service-orientierten System aus Abb. 4.26.

5.3.1. Komposition der Service-Abstraktionen

Durch das Kompositionsverfahren in Def. 5.5 entsteht das Petri-Netz in Abbildung 5.8. Die Stellen, die mit $init_p$, wobei p für den Namen der aufgerufenen Prozedur steht, wird mit dem i_p verknüpft. Die Austrittspunkte werden analog verknüpft. Die Stellen $init_p$ und i_p verschmelzen dabei zur Stelle i_p und die Stellen ret_p und r_p verschmelzen zu r_p.

Behauptung 1. *Die (P,P)-PRS-basierte Abstraktion des Service-orientierten Systems in Abbildung 4.26 ist deadlockfrei.*

Beweis 5.2. Es muss gezeigt werden, dass alle Ausführungen vom Anfangszustand μ_0 des Petri-Netzes, dass das abstrakte Verhalten des Service-orientierten Systems zeigt, in den Finalzustand μ_f gelangen. Wobei $\mu(q_f) = 1$ und für alle anderen Stellen q des Petri-Netzes gilt: $\mu(q) = 0$.

Schritt 1 Jeder Zustand $\mu_1 \in M_1 \widehat{=} \{\mu : \mu(r_a) = 1, \mu(i_b) + \mu(r_b) + \mu(i_c) + \mu(q_{c1}) + \mu(q_{c2}) + \mu(q_{c3}) + \mu(q_{c4}) + \mu(r_c) \leq 3\}$ erreicht immer den Zustand μ_f.

Schritt 2 Jeder Zustand $\mu_2 \in M_2 \widehat{=} \{\mu : \mu(q_{a7}) = 1, \mu(i_b) + \mu(r_b) \leq 2, \mu(i_c) + \mu(q_{c1}) + \mu(q_{c2}) + \mu(q_{c3}) + \mu(q_{c3}) + \mu(r_c) = 1\}$ erreicht immer einen Zustand μ_1 in M_1.

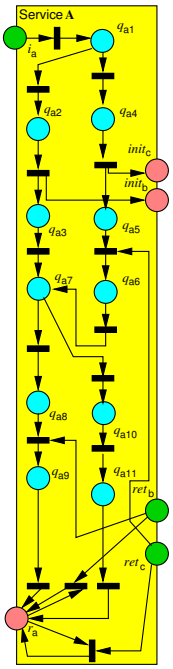


Abbildung 5.5.: Die Petri-Netz-basierte Abstraktion von Service A aus dem Service-orientierten System aus Abb. 4.26.

Schritt 3 Jeder Zustand $\mu_3 \in M_3 \widehat{=} \{\mu : \mu(q_{a1}) = 1, \mu(i_b) + \mu(r_b) = 1, \mu(i_c) + \mu(q_{c1}) + \mu(q_{c2}) + \mu(q_{c3}) + \mu(q_{c4}) + \mu(r_c) = 1\}$ erreicht immer einen Zustand μ_2 in M_2.

Schritt 4 Jeder Zustand μ_0 erreicht immer einen Zustand $\mu_3 \in M_3$.

Mit dem Beweis über die 4 Schritte wird gezeigt, dass der Finalzustand μ_f immer vom Anfangszustand μ_0 aus erreicht wird.

Schritt 1: Betrachten wird der Fall, dass sich eine Marke in r_a befindet. Marken, die sich in den Stellen $i_b, r_b, i_c, q_{c1}, q_{c2}, q_{c3}, q_{c4}$ oder r_c befinden, können durch beliebiges Feuern der Transitionen t_{20} oder t_{22} abgebaut werden. Feuert die Transition t_4 so wird die Marke in r_a nach q_f transportiert. Die Transitionen t_{20} und t_{22} sind nun nicht mehr geladen, dafür aber die Transitionen t_5 und t_6, die mögliche Marken aus r_b und r_c entfernen können, so dass lediglich eine Marke in q_f übrig bleibt. Aus diesem Grund, erreicht jeder Zustand μ_1 den Finalzustand μ_f.

Schritt 2: Die Stelle q_{a7} enthält eine Marke, so dass die Transitionen t_{15} und t_{16} geladen sind. Neben der Marke in q_{a7} befindet sich mindestens 1 Marke in der Stelle i_b oder r_b. Da hier nur die Stelle t_{23} feuern kann, wird der einfachheitshalber angenommen, dass sich die Marke in der Stelle r_b befindet. Analog wird davon ausgegangen, dass sich die Marke

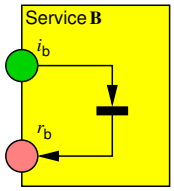


Abbildung 5.6.: Die Petri-Netz-basierte Abstraktion von Service B aus dem Service-orientierten System aus Abb. 4.26.

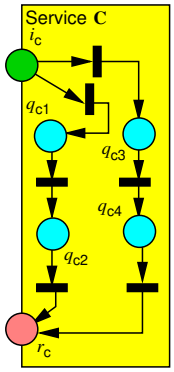


Abbildung 5.7.: Die Petri-Netz-basierte Abstraktion von Service C aus dem Service-orientierten System aus Abb. 4.26.

im Teilnetz von Service C ebenfalls in r_c befindet. In diesem Zustand kann das Netz folgende Feuersequenzen ausführen: $[t_{16}, t_{18}, t_{21}]$ oder $[t_{15}, t_{17}, t_{19}]$. Feuert die Transition t_{17}, so wird eine Marke aus der Stelle r_b entzogen (im Zustand M_1 ist mindestens eine Marke in r_b vorhanden). Bei beiden Feuersequenzen gelangt das Netz in einen Zustand $\mu_1 \in M_1$. Die Stelle r_a enthält eine Marke, r_b kann keine, eine oder zwei Marken enthalten und r_c enthält eine Marke.

Schritt 3: Im diesem Schritt befindet sich eine Marke in q_{a1}, ein Marke in r_b und eine Marke in r_c. Daraus ergeben sich zwei mögliche Feuersequenzen. Die erste Möglichkeit ist $[t_8, t_{10}, t_{12}]$, dabei wird eine weitere Marke durch Feuern von t_{10} in der Stelle i_b erzeugt. Die zweite mögliche Feuersequenz ist $[t_9, t_{11}, t_{13}, t_{14}]$. Hier wird beim Feuern von t_{11} eine Marke in der Stelle i_c erzeugt. Diese Marke wird aber aus dem Teilnetz, dass sich aus dem Service C ergibt, durch das Feuern von t_{13} wieder aus dem Teilnetz entfernt. Dadurch wird ein Zustand erreicht, in dem eine Marke in der Stelle q_{a7} liegt, insgesamt eine oder zwei Marken in den Stellen i_b oder r_b und eine Marke in den Stellen des Teilnetzes von Service C. Die Verteilung der Marken in dem Netz entspricht einem Zustand $\mu_2 \in M_2$.

Schritt 4: Wenn sich das oben genannte Petri-Netz im Anfangszustand befindet, dass heißt, es befindet sich nur eine Marke im Netz und diese Marke liegt in q_0, sodass

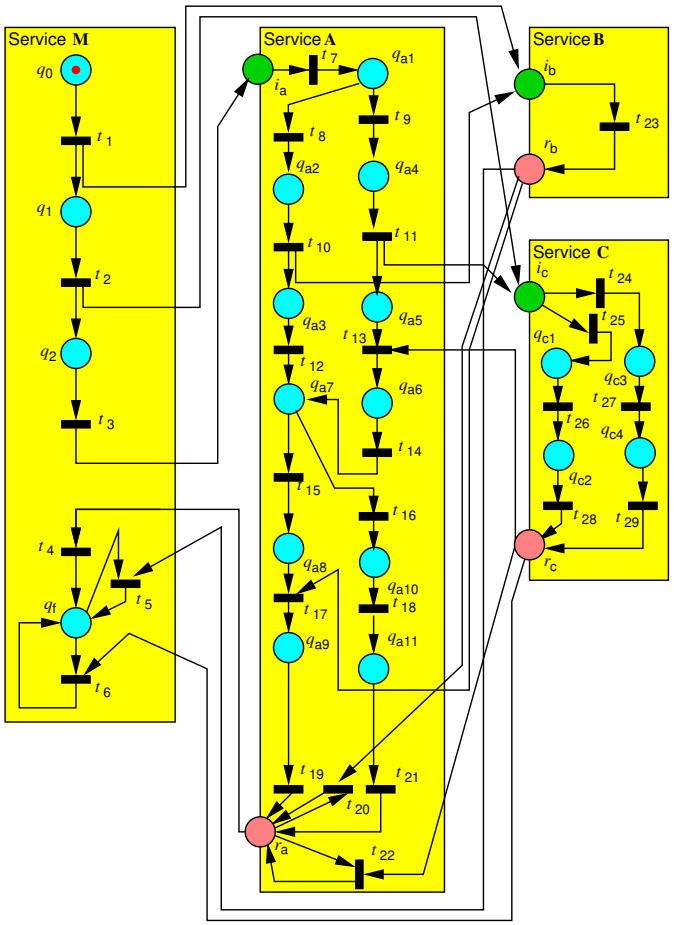


Abbildung 5.8.: Das Petri-Netz welches nach der Komposition der Petri-Netze aus den Abbildungen 5.4-5.7 entsteht.

$\mu(q_0) = 1$ ist, dann ist nur die Transition t_1 geladen. Feuert t_1 so erhält q_1 und q_{i_b} eine Marke. Es sind nun die Transitionen t_{23} im Petri-Netz von Service B und t_2 in Service M geladen. Diese Markierung als Ausgangsmarkierung könnten nun die Transitionen in folgender Reihenfolge feuern $[t_{23}, t_2, t_3, t_4]$, $[t_2, t_{23}, t_3, t_4]$, $[t_2, t_3, t_{23}, t_4]$ oder $[t_2, t_3, t_4, t_{23}]$. Mit Feuern der Transition t_2 bekommt die Stelle i_c eine Marke, die zur Folge hat, dass auch die Transitionen t_{24} und t_{25} geladen sind. Da beim Feuern dieser Transitionen die Marke in i_c lediglich nach r_c transportiert wird und das Feuern der Transitionen t_{23}, t_2, t_3 und t_4 nicht beeinflusst, wird aus Platzgründen auf die Angabe der verschiedenen Kombinationen beim Feuern verzichtet. Eine Marke in r_c führt nicht dazu, dass ein Stelle im Nachbreich von r_c geladen ist. Die einzigen Transitionen, die nun geladen sind, sind die Transitionen t_8 und t_9. Das Netz enthält damit 3 Marken, wobei eine Marke in der Stelle q_{a7} liegt, eine Marke liegt in der Stelle i_b oder r_b, abhängig davon, ob t_{23} bereits gefeuert hat oder nicht und eine Marke liegt in i_c, q_{c1}, q_{c2}, q_{c3} oder q_{c4} abhängig davon, ob die

Transitionen t_{24} bis t_{29} bereits gefeuert haben. Damit wird der Zustand μ_3 erreicht, der sich in M_3 befindet.

Damit wurde gezeigt, dass das Petri-Netz immer vom Anfangszustand μ_0 in den Finalzustand μ_f gelangen kann. Somit ist das vorliegende Petri-Netz deadlockfrei.

5.3.2. Zusammenfassung

Das durch die Abstraktion entstandene Petri-Netz (Abb. 5.8) ist deadlockfrei. Alle Ausführungen vom Anfangszustand μ_0 gelangen zum Finalzustand μ_f. Im Kapitel 4 wird mit der dort eingeführten operationalen Semantik gezeigt, dass das betrachtete Service-orientierte System in Abbildung 4.26 in einem Deadlock endet. Es kommt zu einer falsch positiven Aussage. Das Service-orientierte System enthält einen Deadlock, während die Petri-Netz-basierte Abstraktion keinen Deadlock enthält und das System fälschlicherweise als deadlockfrei einstuft.

Wie im Bereich der Protokollverletzungen, gibt es auch bei der Deadlockanalyse falsch positive Aussagen bei der Nutzung von Petri-Netz-basierten Ansätzen.

5.4. Zusammenfassung und Diskussion

In diesem Kapitel wurde das Abstraktionsverfahren von Service-orientierten Systemen zu Petri-Netzen gezeigt. Im Anschluss wurde am konkreten Beispiel das Abstraktionsverfahren durchgeführt. Anhand des Service-orientierte System aus Kapitel 4 wurde gezeigt, dass nicht alle auftretenden Deadlocks erkannt werden können. Die Anwendung des Verfahrens führt zu einem Petri-Netz, dass bei allen Ausführungen zum Finalzustand gelangt, obwohl ein Deadlock vorhanden sein kann.

Dieses Gegenbeispiel zeigt die Grenzen Petri-Netz basierter Ansätze auf. Während der Abstraktion geht im Beispiel von Beweis 5.2 die Information über den Aufrufkontext verloren. Da Transitionen in Petri-Netzen bzw. der parallele Operator „||“ kommutativ ist, ist die Transition immer geladen und kann damit feuern, obwohl der Aufruf nicht im Kontext des Aufgerufenen steht.

Übertragen auf die eingangs beschriebene Abstraktion eines konkreten Systems C zu einem abstrakten System A in Abbildung 5.1 wird keine Obermenge über alle möglichen Läufe eines konkreten Systems C, die ausreichend hinsichtlich des Erreichbarkeitsproblems und anderen Sicherheitsbedingungen ist, gebildet.

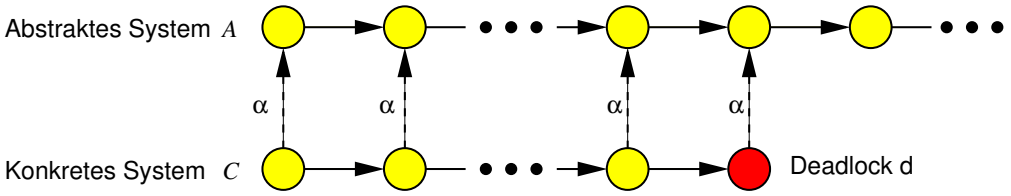


Abbildung 5.9.: Abstraktion bei denen der Deadlock nicht im abstrakten System konserviert wird.

Dieses Problem wird in Abbildung 5.9 gezeigt. d ist ein Deadlock im konkreten System C. Während die Abstraktion $\alpha(d)$ kein Deadlock im abstrakten System A ist. Die Abstraktion $\alpha(d)$ eines Deadlock d von C hat aber eventuell einen Nachfolger in A, gezeigt in [54]. Möglicherweise ist A aber auch frei von Deadlocks, vgl. dazu mit dem Gegenbeispiel 4.26 in diesem Kapitel.

Aus diesem Grund können Abstraktion unter Umständen zu vermeintlich unechten Gegenbeispielen führen. Daher muss sichergestellt, dass gilt: $\{\alpha(\mu) : \mu \text{ deadlock in } C\} \subseteq \{\mu : \mu \text{ deadlock in } A\}$.

Daraus lässt sich folgern, dass die Analyse basierend auf Petri-Netzen nicht ausreicht. Bisherige Deadlockanalyse-Verfahren stoßen hier an ihre Grenzen und kommen zu falschen Aussagen bezüglich der Deadlockfreiheit von Programmen.

Aus diesem Grund werden im kommenden Kapitel 6 die Erhaltung von Deadlocks bei (G,G)-PRS-basierten Ansätzen untersucht und diskutiert.

6. Deadlockanalyse mit PRS-basiertem Ansatz

In Kapitel 5 wurden die Grenzen Petri-Netz-basierter Ansätze gezeigt. Das konkrete Verhalten des Service-orientierten Systems konnte durch die Petri-Netz-Abstraktion nicht dargestellt werden. Bei der Deadlockanalyse traten falsch positive Ergebnisse auf. Der Deadlock des im Kapitel 4 präsentierten Beispiels, konnte durch das abstrakte Verhalten basierend auf Petri-Netzen nicht dargestellt werden.

Daher wird in diesem Kapitel das abstrakte Verhalten basierend auf (G,G)-PRS-basierten Abstraktionen untersucht. Dabei soll geprüft werden, ob ein (G,G)-PRS-basierter Ansatz Deadlocksituationen in einem Service-orientierten Softwaresystem konservieren kann. Das Service-orientierte System kann (rekursive) synchrone und asynchrone Prozedur- bzw. Funktionsaufrufe beinhalten.

6.1. (G,G)-PRS-basierte Abstraktion von Programmen

Im Folgenden wird das (G,G)-PRS-Abstraktionsverfahren beschrieben. Es wird gezeigt, wie durch Anwendung einer (G,G)-PRS-Abstraktionsfunktion aus der operationalen Semantik $[\![\Pi]\!]_{konkret}$ die (G,G)-PRS-basierte Abstraktion $[\![\Pi]\!]_{prs}$ eines Programms mit dem Quellcode Π gewonnen werden kann, (vgl. Abbildung 6.8) und wie dieses Verfahren auf Services angewendet werden kann.

Bemerkung 6.1. *Das Abstraktionsverfahren eines konkreten Programms zu einem (G,G)-PRS nach [11] wird im Grundlagenkapitel unter dem Abschnitt 3.3.1 erläutert. Der Unterschied hier ist, dass bei Aufrufen asynchroner Prozeduren das Future des Aufrufers bei der aufgerufenen Prozedur mitgeführt wird.*

Damit ergibt sich folgende Definition für die (G,G)-PRS-basierte Abstraktion eines Programms:

Definition 6.1 ((G,G)-PRS-basierte Abstraktion eines Programms)**.** Die (G,G)-PRS-basierte Abstraktion eines Programms Π der Beispielsprache $\mathcal{XYZ}$ wird durch das Zustandsübergangssystem $[\![\Pi]\!]_{prs}$ beschrieben. Dieses Zustandsübergangssystem ergibt

sich aus der Anwendung der (G,G)-PRS-Abstraktionsfunktion α_{prs} auf die Regelmenge $\rightarrow_{konkret}$ eines Zustandsübergangssystem $[\![\Pi]\!]_{konkret}$, welches die konkrete (operationale) Semantik des Programms Π beschreibt.

Damit ergibt sich das Zustandsübergangssystem $[\![\Pi]\!]_{prs}$ der (G,G)-PRS-basierte Abstraktion durch die Anwendung der (G,G)-PRS-Abstraktionsfunktion:

Definition 6.2 ((G,G)-PRS-Abstraktionsfunktion)**.** Sei π ein service-orientiertes System und VAR die Menge der Futures in π, wobei angenommen wird, dass die Future-Namen eindeutig sind. Die Funktion $proc : NODE \rightarrow PROC$ sei die Funktion, die zu jedem $q \in NODE$ die dazugehörige Prozedur ermittelt. Sei $(q, \sigma) \in SFRAME$, so dass $p \hat{=} proc(q)$ asynchron ist und $e \in PEX(SFRAME)$. Das aufrufende Future f von (q, σ) in e ist $future((q, \sigma), e) \hat{=} f$ mit $(q', \sigma') \in e$ mit $\sigma'(f) = (f, \sigma(p))$. Die (G,G)-PRS-Abstraktionsfunktion $\alpha_{prs} : PEX(SFRAME) \rightarrow ((PEX(NODE)) \cup (PEX(NODE) \times VAR))$ ist definiert durch $\alpha_{prs}(e) = \alpha'_{prs}(e, e)$ wobei

- $\alpha'_{prs}((q, \sigma), e) \hat{=} (q, future((q, \sigma), e))$, falls $proc(q)$ asynchron ist,
- $\alpha'_{prs}((q, \sigma), e) \hat{=} q$, falls $proc(q)$ synchron ist,
- $\alpha'_{prs}(e_1.e_2, e) \hat{=} \alpha'_{prs}(e_1, e).\alpha'_{prs}(e_2, e)$ und
- $\alpha'_{prs}(e_1||e_2, e) \hat{=} \alpha'_{prs}(e_1, e)||\alpha'_{prs}(e_2, e)$,

wobei e_1, e_2 und e sind prozess-algebraische Ausdrücke über $PEX(SFRAME)$ und $q \in NODE$ ist.

Durch die Anwendung der (G,G)-PRS-Abstraktionsfunktion α_{prs} auf die Regeln der konkreten Semantik $[\![\Pi]\!]_{abstrakt}$ ergibt sich damit folgende Definition für (G,G)-PRS-Zustandsübergangssysteme:

Definition 6.3 ((G,G)-PRS-Zustandsübergangsystem)**.** Ein (G,G)-PRS-Zustandsübergangssystem wird definiert durch ein Quadrupel $[\![\Pi]\!]_{prs} = (Z_{prs}, \rightarrow_{prs}, q_{0_{prs}}, F)$, dabei sei

- $Z_{prs} \subseteq PEX((NODE) \times (NODE, VAR))$ eine Menge von Ausdrücken,
- $q_{0_{prs}} \hat{=} (q_0) \in NODE$ der Startzustand,
- F ist die einelementige Menge mit dem Finalzustand $q_{f_{prs}} \in NODE$,
- $\rightarrow_{prs} \subseteq PEX(NODE \times (NODE, VAR)) \times PEX(NODE \times (NODE, VAR))$ eine Zustandsübergangsrelation, die sich durch Anwendung von α_{prs} ergibt, siehe Tab. 6.1.

Die Anwendung der Abstraktionsfunktion α_{prs} auf die Inferenzregeln der operationalen Semantik $[\![\Pi]\!]_{konkret}$ wird in Tabelle 6.1 gezeigt. In der ersten Spalte werden die Kontrollstrukturen aus Quellcode Π des konkreten Programms aufgeführt. Die zweite Spalte beinhaltet jeweils die Inferenzregel der konkreten Semantik $[\![\Pi]\!]_{konkret}$, die das Verhalten der jeweiligen Kontrollstruktur darstellt. In der dritten Spalte werden die Regeln des Zustandssystems $[\![\Pi]\!]_{prs}$ der (G,G)-PRS-basierten Abstraktion gezeigt, die sich aus der Anwendung von α'_{prs} und α_{prs} auf die Inferenzregel der konkreten Semantik $[\![\Pi]\!]_{konkret}$ ergeben. Das Ergebnis der Anwendung der Abstraktionsfunktion α_{prs} unterscheidet sich im Vergleich zur Petri-Netz-Abstraktionsfunktion (Kap. 5) nur in der dritten und vierten Zeile (siehe Markierung). Bei dem Aufruf einer synchronen Prozedur p wird der Programmpunkt q' nach dem synchronen Aufruf bei Programmpunkt q aus der Inferenzregel abstrahiert, sodass der Ausdruck $i_p.q'$ entsteht. Der Programmpunkt i_p ist dabei der Eintrittsprogrammpunkt der synchronen Prozedur p im Originalprogramm Π. Bei der Rückkehr der synchronen Prozedur, wird der zum Ausdruck $r_p.q'$ abstrahiert. In der vierten Zeile, wird im Gegensatz zu den Petri-Netz-Abstraktionen der Aufrufkontext in Form der Variablen des Futures in φ, hier f, in die (G,G)-PRS-basierte Abstraktion übernommen.

Die grafische Darstellung der Kaktuskeller bei Ausführung der (G,G)-PRS-Regeln ist im Grundlagenkapitel unter der Tabelle 3.1 zu finden.

Die (G,G)-PRS-basierte Abstraktion für Services wird im nächsten Abschnitt erläutert.

6.2. (G,G)-Abstraktion von Services und deren Komposition

In diesem Abschnitt wird analog dem vorangegangen Kapiteln diskutiert, wie die (G,G)-(PRS)-Abstraktion auf Services übertragen und die Komposition der Abstraktionen analog den vorangegangenen Kapiteln durchgeführt werden können.

Die (G,G)-PRS-basierten Abstraktionen für Services werden nach dem gleichen Prinzip wie die Petri-Netz-Abstraktionen erstellt. Zur Darstellung des abstrakten Verhalten auf Basis von (G,G)-PRS von Services wird die Tabelle zur Abstraktion der Kontrollstrukturen in monolithischen Programmen (Tab. 6.1) erweitert. Die Erweiterung beinhaltet auch hier, wie bei der in Kapitel 5 vorgestellten Petri-Netz-Abstraktion, die Abstraktion von Service-übergreifenden Prozedur- und Funktionsaufrufen. Die Kennzeichnung externer Prozedur- oder Funktionsaufrufe einer Prozedur oder Funktion p werden von den Regeln der operationalen Semantik übernommen und ebenfalls mit $init_p$ und ret_p gekennzeichnet.

Diese Platzhalter werden bei der Komposition der Services über die bereits bekannte Platzhalterfunktion $\mathcal{D}$, die die Services mit Angebotsschnittstellen zur Verfügung stellen,

Kontrollstruktur	Inferenzregel aus $[\![\Pi]\!]_{konkret}$	Anwendung von α_{prs}
q : $x := e;$ q' : $\cdots$	$\frac{}{(q,\sigma) \to (q', \sigma'\vert_x^{eval_A^\sigma(e)})}$	$\frac{}{q \to q'}$, falls $proc(q)$ synchron $\frac{}{(q,x) \to (q',x)}$ für alle Futures $x \in VAR$, falls $proc(q)$ asynchron

q_1 : **if** e{
q_2 : $\cdots$
q_3 : $\cdots\}$
else{
q_4 : $\cdots$
q_5 : $\cdots\}$
q_6 : $\cdots$
	$\frac{eval_B^\sigma(e) = true}{(q_1,\sigma) \to (q_2,\sigma)}$
	$\frac{eval_B^\sigma(e) = false}{(q_1,\sigma) \to (q_4,\sigma)}$
	$\frac{}{(q_3,\sigma) \to (q_6,\sigma)}$
	$\frac{}{(q_5,\sigma) \to (q_6,\sigma)}$
		Falls $proc(q_1)$ synchron:
		$\frac{}{q_1 \to q_2}$
		$\frac{}{q_1 \to q_4}$
		$\frac{}{q_3 \to q_6}$
		$\frac{}{q_5 \to q_6}$
		Falls $proc(q_1)$ asynchron für alle Futures $x \in VAR$:
		$\frac{}{(q_1,x) \to (q_2,x)}$
		$\frac{}{(q_1,x) \to (q_4,x)}$
		$\frac{}{(q_3,x) \to (q_6,x)}$
		$\frac{}{(q_5,x) \to (q_6,x)}$

Synchrone Prozedur p q : $p;$ q' : $\cdots$ $p()${ i_p : $\cdots$ r_p : **return;**$\}$	$\frac{}{(q,\sigma) \to (i_p, \sigma').(q',\sigma)}$, $\frac{}{(r_p,\sigma').(q',\sigma) \to (q',\sigma)}$	Falls $proc(q)$ synchron: $\frac{}{q \to i_p.q'}$ $\frac{}{r_p.q' \to q'}$ Falls $proc(q)$ asynchron für alle Futures $x \in VAR$: $\frac{}{(q,x) \to i_p.(q',x)}$ $\frac{}{r_p.(q,x)' \to (q',x)}$
Asynchrone Prozedur p $a\{\cdots$ $future f;$ $\cdots$ q : $f := p();$ q' : $\cdots$ q'' : **return;**$\}$ $p()${ i_p : $\cdots$ r_p : **return;**$\}$	$\frac{}{(q,\sigma) \to (q', \sigma'\vert_f^{(p,\Phi)}) \vert\vert (\bar{q}, \sigma''\vert_p^\Phi)}$, $\frac{\sigma(f) = (p, \sigma'(p))}{(q'',\sigma)\vert\vert(\bar{q}',\sigma') \to (q'',\sigma)}$	Falls $proc(q)$ synchron: $\frac{}{q \to q' \Vert (i_p, f)}$ $\frac{}{q'' \Vert (r_p, f) \to q''}$ Falls $proc(q)$ asynchron für alle Futures $x \in VAR$: $\frac{}{(q,x) \to (q',x) \Vert (i_p,f)}$ $\frac{}{(q'',x) \Vert (r_p,f) \to (q'',x)}$
Synchronisation q : **sync** $f;$ q' : $\cdots$ $p()${ i_p : $\cdots$ r_p : **return;**$\}$	$\frac{\sigma(f) = (q, \sigma'(p))}{(r_p,\sigma')\vert\vert(q,\sigma) \to (q',\sigma)}$	Falls $proc(q)$ synchron: $\frac{}{(r_p,f) \Vert q, \to q'}$ Falls $proc(q)$ asynchron für alle Futures $x \in VAR$: $\frac{}{(r_p,f) \Vert (q,x), \to (q',x)}$

Tabelle 6.1.: (G,G)-PRS-basierte Abstraktion: Anwendung der (G,G)-PRS-Abstraktionsfunktion α_{prs} auf die Inferenzregeln der operationalen Semantik $[\![\Pi]\!]_{konkret}$.

aufgelöst. Die Platzhalterfunktion $\mathcal{D}$ in Tabelle 6.2 wird definiert über $\mathcal{D}(init_p) = i_p$ und $\mathcal{D}(ret_p) = r_p$ für synchrone Prozeduren sowie $\mathcal{D}(init_p, \varphi) = (i_p, \varphi)$ und $\mathcal{D}(ret_p, \varphi) = (r_p, \varphi)$ für asynchrone Prozeduren in den Angebotsschnittstellen.

Bemerkung 6.2. *Auch bei der (G,G)-PRS-basierte Abstraktion werden die im Kapitel 4 in der operationalen Semantik eingeführten zusätzlich Programmpunkte $s_p \in NNODE_S$ und $re_p \in NNODE_S$ einer aufgerufenen Prozedur oder Funktion eines Service S der abstrakten Semantik verworfen, da diese Programmpunkte nur Hilfsmittel zur Darstellung des konkreten Verhaltens, im Speziellen der Übertragung von Parametern, dienen.*

Nach der Einführung der Platzhalter muss für die (G,G)-PRS-basierte abstrakte Darstellung von Services die Semantik von Prozeduren in den Angebotsschnittstellen definiert werden:

Definition 6.4 ((G,G)-PRS-basierte Abstraktion einer Prozedur in Angebotsschnittstellen)**.** Sei I_S eine Angebotsschnittstelle eines Service S mit der Prozedur oder Funktion $p \in I_S$ mit einem eindeutigen Eintritts-Programmpunkt $i_p \in NODE_S^{I_A}$, die erste Zeile Programmcode im Rumpf von p, und Austritts-Programmpunkt $r_p \in NODE_S^{I_A}$, die *return*-Anweisung der Prozedur p.

Die Menge der Platzhalter $DNODE_{S_p}$ beinhaltet die Elemente $init_p$ und ret_p.

Die (G,G)-PRS-basierte Abstraktion von S bezüglich einer Prozedur p einer Angebotsschnittstelle I_S ist das Zustandsübergangssystem

$$[\![S^{I_{S_p}}]\!]_{prs} = (Z_{I_{S_{pprs}}}, q_{0_{I_{S_{pprs}}}}, \rightarrow_{I_{S_{pprs}}}, F_{I_{S_{pprs}}}, \mathcal{D}_{I_{S_{pprs}}}), \text{ wobei:}$$

- $Z_{I_{S_{pprs}}} \subseteq PEX(NODE_S \cup DNODE_{S_p} \cup NODE_S \times VAR \cup DNODE_{S_p} \times VAR)$ eine Menge von Zuständen,
- $q_{0_{I_{S_{pprs}}}} = i_p^{I_S}$ der Anfangszustand bei synchronen Prozeduren und $q_{0_{I_{S_{pprs}}}} = (i_p^{I_S}, \varphi)$ der Anfangszustand bei asynchronen Prozeduren, wobei $i_p^{I_S} \in NODE_S^{I_S}$ und $\varphi \in VAR$,
- $\rightarrow_{I_{S_{pprs}}} \subseteq Z_{I_{S_{pprs}}} \times Z_{I_{S_{pprs}}}$ ist die Zustandsübergangsrelation auf Grundlage der Ableitungsrelationen der operationalen Semantik durch Anwendung von α_{prs} (Tab. 6.1 und 6.2),
- $F_{I_{S_{pprs}}} = \{r_p^{I_S}\}$ wobei $r_p^{I_S}$ Endzustand bei synchronen Prozeduren bzw. $F_{I_{S_{pprs}}} = \{(r_p^{I_S}, \varphi)\}$ bei asynchronen Prozeduren ist, wobei $r_p^{I_S} \in NODE^{I_S}$ und $\varphi \in VAR$ und,

Kontrollstruktur	Inferenzregel aus $[\![\Pi]\!]_{konkret}$	Anwendung von α_{prs}
Synchrone Prozedur p q : p; q' : $\cdots$ $p()\{$ i_p : $\cdots$ r_p : **return;**$\}$	$\frac{}{(q,\sigma) \to (in_p^{R_A}, \sigma_i^{dy}).(q',\sigma)}$, $(ret_p^{R_A}, \sigma_r^{dy}).(q',\sigma) \to (q', \sigma\vert_y^{eval^{\sigma_r^{dy}}(rval)})$	Falls $proc(q)$ synchron: $\frac{}{\overline{q \to init_p.q'}}$ $\overline{ret_p.q' \to q'}$ Falls $proc(q)$ asynchron für alle Futures $x \in VAR$: $\frac{}{\overline{(q,x) \to init_p.(q',x)}}$ $\overline{ret_p.(q',x) \to (q',x)}$
Asynchrone Prozedur p $a\{\cdots$ $future f;$ $\cdots$ q : $f := p();$ q' : $\cdots$ q'' : **return;**$\}$ $p()\{$ i_p : $\cdots$ r_p : **return;**$\}$	$\frac{}{(q,\sigma) \to (q', \sigma'\vert_f^{(p,n)}) \vert\vert (init_p^{R_A}, \sigma_i^{dy}\vert_p^n)}$, $\frac{\sigma(f) = (p, \sigma_r^{dy}(p))}{(q'',\sigma) \parallel (ret_p^{R_A}, \sigma_r^{dy}) \to (q'',\sigma)}$	Falls $proc(q)$ synchron: $\frac{}{\overline{q \to q' \Vert (init_p, f)}}$ $\overline{q'' \Vert (ret_p, f) \to q''}$ Falls $proc(q)$ asynchron für alle Futures $x \in VAR$: $\frac{}{\overline{(q,x) \to (q',x) \Vert (init_p, f)}}$ $\overline{(q'',x) \Vert (ret_p, f) \to (q'',x)}$
Synchronisation q : **sync** f; q' : $\cdots$ $p()\{$ i_p : $\cdots$ r_p : **return;**$\}$	$\frac{\sigma(f) = (p, \sigma_r^{dy}(p))}{(q,\sigma) \parallel (ret_p^{R_A}, \sigma_r^{dy}) \to (q',\sigma)}$	Falls $proc(q)$ synchron: $\frac{}{\overline{(q,x) \Vert (ret_p, f) \to (q',x)}}$ Falls $proc(q)$ asynchron für alle Futures $x \in VAR$: $\frac{}{\overline{(q,x) \Vert (ret_p, f) \to (q',x)}}$

Tabelle 6.2.: (G,G)-PRS-basierte Abstraktion von Service-übergreifenden Prozeduraufrufen. Die Platzhalterfunktion $\mathcal{D}$ ist definiert durch $\mathcal{D}(init_p) = i_p$ und $\mathcal{D}(ret_p) = r_p$.

- $\mathcal{D}_{I_{S_{pprs}}} : ((DNODE_{S_p})) \rightarrow (NODE_S^{I_S})$ bei synchronen Prozeduren bzw. $\mathcal{D}_{I_{S_{pprs}}} : DNODE_{S_p} \times VAR \rightarrow NODE_S^{I_S} \times VAR$ bei asynchronen Prozeduren die Platzhalterfunktion.

Definition 6.5 ((G,G)-PRS-basierte Abstraktion der Prozedur main)**.** Die (G,G)-PRS-basierte Abstraktion der main-Prozedur m eines Klientenservice K ist definiert durch das Zustandsübergangssystem $[\![K_m]\!]_{prs} = (Z_{K_{m_{prs}}}, q_{0_{K_{prs}}}, \rightarrow_{K_{m_{prs}}}, F_{K_{m_{prs}}}, \mathcal{D}_{K_{m_{prs}}})$, wobei:

- $Z_{K_{m_{prs}}} \subseteq PEX(NODE_K \cup DNODE^{R_K} \cup DNODE^{R_K} \times VAR)$ eine Menge von Zuständen,
- $q_{0_{K_{prs}}} = q_0$ der Anfangszustand,
- $\rightarrow_{K_{m_{prs}}} \subseteq Z_{K_{m_{prs}}} \times Z_{K_{m_{prs}}}$ ist die Zustandsübergangsrelation auf Grundlage der Ableitungsrelationen der operationalen Semantik durch Anwendung von α_{prs} (Tab. 6.1 und 6.2),
- $F_{K_{m_{prs}}} = \{q_f\}$, wobei q_f der Endzustand (Programmpunkt mit der *return*-Anweisung im Rumpf von m) und
- $\mathcal{D}_{K_{m_{prs}}}$ nicht definiert ist.

Daraus ergibt sich das (G,G)-PRS-basierte Zustandsübergangssystem $[\![S]\!]_{prs}$:

Definition 6.6 ((G,G)-PRS-basierte Abstraktion eines Services)**.** Sei S ein Service mit Angebotsschnittstellen $I_{S_1}, \ldots, I_{S_n}$ und Nutzungsschnittstellen $R_{S_1}, \ldots, R_{S_m}$ mit $n, m \in \mathbb{N}$ und einer Implementierung mit Programmpunkten $NODE_S$. Die (G,G)-PRS-basierte Abstraktion von S ist die Menge aller (G,G)-PRS-basierten Zustandsübergangssysteme ihrer Angebotsschnittstellen $[\![S]\!]_{prs} = [\![S^{I_{S_1}}]\!]_{prs} \cup \cdots \cup [\![S^{I_{S_n}}]\!]_{prs}$ mit $n \in \mathbb{N}$.

Die Menge der Zustandsübergangssysteme $[\![S^{I_S}]\!]$ einer Angebotsschnittstelle I_S ergibt sich aus der Menge der Zustandsübergangssysteme aller Prozeduren oder Funktionen $p_1, \ldots, p_n$ der Angebotsschnittstellen I_S: $[\![I_S]\!]_{prs} = [\![I_{S_{p_1}}]\!]_{prs} \cup \cdots \cup [\![I_{S_{p_n}}]\!]_{prs}$ mit $n \in \mathbb{N}$.

$[\![I_S]\!]_{prs} = (Z_{I_{S_{prs}}}, Q_{0_{prs}}, \rightarrow_{S_{prs}}, F_{S_{prs}}, \mathcal{D}_{prs})$, wobei:

- $Z_{I_{S_{prs}}} \subseteq PEX(DNODE_S \cup NODE_S \cup DNODE_S \times VAR \cup NODE_S \times VAR)$ eine Menge von Kaktuskellern,
- $Q_{0_{prs}} = \{i_p^{I_S} : p \in I_S \text{ und } p \text{ synchron}\} \cup \{(i_p^{I_S}, \varphi) : p \in I_S \text{ und } p \text{ asynchron}\}$ die Anfangszustände aller Prozeduren p in I_S und $\varphi \in VAR$,

- $\rightarrow_{S_{prs}} = \bigcup_{i=1}^{n} \rightarrow_{I_{S_{p_{i_{prs}}}}}$ ist die Menge der Regeln der Zustandsübergangsrelationen aller Prozeduren und Funktionen p_i in der Angebotsschnittstelle I_S,

- $F_{S_{prs}} = \{(r_p^{I_S}) : p \in I_S, \text{und } p \text{ ist synchron}\} \cup$
  $\{(r_p^{I_S}, \varphi) : p \in I_S,\ \varphi \in VAR \text{ und } p \text{ ist asynchron}\}$
  der Endzustände aller Prozeduren p in I_S,

- $\mathcal{D}_{prs} = \bigcup_{i=1}^{n} \mathcal{D}_{I_{S_{p_{i_{prs}}}}}$ die Menge der Platzhalterfunktionen.

Liegen die (G,G)-PRS-basierten Abstraktionen aller Services des Service-orientierten Systems vor, können diese nach der Definition 6.7 miteinander verknüpft werden.

Definition 6.7 (Komposition der (G,G)-PRS-basierten Abstraktionen)**.** Seien $S_1, \ldots, S_n$ Services mit Angebotsschnittstellen I_i und Nutzungsschnittstellen R_i und $SoS[S_1, \ldots, S_n]$ ein Service-orientiertes System mit den Services $S_1, \ldots, S_n$ sowie
$[\![S_i]\!]_{prs} \widehat{=} (Z_{S_{i_{prs}}}, Q_{0_{S_{i_{prs}}}}, \rightarrow_{S_{i_{prs}}}, F_{S_{i_{prs}}}, \mathcal{D}_{S_{i_{prs}}})$ mit $i = 1, \cdots, n$ und $n \in \mathbb{N}$.

Die Komposition der (G,G)-PRS-basierten Service-Abstraktionen ist eine kombinierte (G,G)-PRS-basierte-Abstraktion $SoS[[\![S_1]\!]_{prs}, \ldots, [\![S_n]\!]_{prs}) \widehat{=} (Z_{prs}, [\![\rightarrow]\!]_{prs}, Q_{0_{prs}}, Q_{f_{prs}})$ mit

- $Z_{prs} = Z_{sync} \cup \{(q, f) : (q, \varphi) \in Z_{async}, (init_{proc(q)}, f) \in Z_{all}\}$, wobei
  $Z_{all} = Z_{S_{1_{prs}}} \cup \cdots \cup Z_{S_{1_{n}prs}}$, $Z_{sync} = \{q \in Z_{all} : proc(q) \text{ synchron}\}$ und
  $Z_{async} = \{q \in Z_{all} : proc(q) \text{ asynchron}\}$

- $[\![\rightarrow]\!]_{prs} \widehat{=} \mathcal{D}_1([\![\rightarrow_1]\!]_{prs}) \cup \cdots \cup \mathcal{D}_n([\![\rightarrow_n]\!]_{prs})$, wobei
  $\mathcal{D}(q) \widehat{=} \begin{cases} \mathcal{D}_j(init_p), & \text{falls } p \in R_i \cap I_j,\ p \text{ synchron und } R_i \rightarrow I_j \in SoS \\ \mathcal{D}_j(init_p, f), & \text{falls } (init_p, f) \in Z_{S_{i_{prs}}},\ p \in R_i \cap I_j,\ p \text{ asynchron und } R_i \rightarrow I_j \in SoS \\ \mathcal{D}_j(ret_p), & \text{falls } p \in R_i \cap I_j,\ p \text{ synchron und } R_i \rightarrow I_j \in SoS \\ \mathcal{D}_j(ret_p, f), & \text{falls } (ret_p, f) \in Z_{S_{i_{prs}}}, p \in R_i \cap I_j\ p \text{ asynchron und } R_i \rightarrow I_j \in SoS \\ q, & \text{sonst} \end{cases}$

- und $Q_{0_{prs}}$ sowie $Q_{f_{prs}}$ sind die Mengen der Anfangszustände und Endzustände aller Klienten im Service-orientierten System.

Durch die Komposition der (G,G)-PRS-basierten Service-Abstraktionen entsteht eine (G,G)-PRS-basierte Abstraktion, die das abstrakte Verhalten des zugrundeliegenden Service-orientierten Systems beschreibt.

In diesem Abschnitt wurde die (G,G)-PRS-Abstraktionsfunktion α_{prs} zur Erstellung des (G,G)-PRS-basierten abstrakten Verhaltens aus dem konkreten Verhalten eines monolithischen Programms vorgestellt. Das Abstraktionsverfahren wurde um die aus Kapitel 4 Dummy-Programmpunkte, die als Platzhalter fungieren, erweitert, um auch die

Darstellung des (G,G)-PRS-basierten abstrakten Verhaltens von Services zu beschreiben. Um das (G,G)-PRS-basierte abstrakte Verhalten eines Service-orientierten Systems zu beschreiben, wurde ein Kompositionsverfahren (Def. 6.7) definiert. Im nächsten Abschnitt wird anhand des Beispiels 4.26 aus Kapitel 5 gezeigt, wie die Abstraktion und Komposition auf Basis der Definitionen durchgeführt werden. Anschließend wird gezeigt, dass der im Kapitel 4 gezeigte Deadlock erkannt wird und keine falsch positiven Ergebnisse auftreten können.

6.3. Beispiel einer Deadlockanalyse

Das Beispiel in Abbildung 4.26 wird in diesem Abschnitt erneut betrachtet, hier im speziellen der Lauf, der im Abschnitt 5.3 zu einem Deadlock führt. Dazu werden im ersten Schritt die (G,G)-PRS-basierten Abstraktionen der einzelnen Services erstellt. Im zweiten Schritt erfolgt die Komposition der (G,G)-PRS-basierten Abstraktionen zu einer (G,G)-PRS-basierten Abstraktion des Service-orientierten Systems. Im letzten Schritt wird gezeigt, dass der Lauf in der (G,G)-PRS-basierten Abstraktion wie auch im Lauf des Zustandsübergangssystem der operationalen Semantik des Service-orientierten Systems in einem Deadlock endet.

Im folgenden werden die (G,G)-PRS-basierten Abstraktionen der einzelnen Service des Service-orientierten Systems aus Abbildung 4.26 vorgestellt.

Service M ist der Klientenservice vom Service-orientierten System in Abbildung 4.26. Die (G,G)-basierte Abstraktion vom Klientenservice ist das Zustandsübergangssystem $[\![M]\!]_{prs} = (Z_{M_{m_{prs}}}, q_0, \rightarrow_{M_{m_{prs}}}, \{q_f\}, \mathcal{D}_{M_{m_{prs}}})$ mit:

- $Z_{M_{m_{prs}}} = PEX(NODE_M \cup DNODE^{R_M} \cup DNODE^{R_M} \times VAR)$,
- $q_0 \in NODE_M$ der erste Programmpunkt in *main* des Klienten M,
- $\rightarrow_{M_{m_{prs}}}$ in Abbildung 6.1 und
- $q_f \in NODE_M$ der letzte Programmpunkt in *main* des Klienten M ist.

Die Platzhalterfunktion $\mathcal{D}_{M_{m_{prs}}}$ ist nicht definiert, da der Klient M keine Angebotsschnittstellen zur Verfügung stellt.

In Abbildung 6.1 werden neben den Regeln des Zustandsübergangssystems $[\![M]\!]_{prs}$ auch die Regeln des Zustandsübergangssystems $[\![M]\!]_{konkret}$ der operationalen Semantik aus Abbildung 4.22 gezeigt. Die Inferenzregel $\frac{}{(q_m,\sigma_0)\rightarrow(q_0,\sigma|_x^{eval^{\sigma_0}(x)})}$ fällt weg, da eine Speicherung möglicher Eingaben nicht in der abstrakten Semantik betrachtet wird. Auf die

übrigen Regeln wird die (G,G)-PRS-Abstraktionsfunktion α_{prs} angewendet, so dass die Regeln in der zweiten Spalte entstehen.

Regelmenge des konkreten Verhaltens von Service M, Fig. 4.21	(G,G)-PRS-Regeln $\rightarrow_{K_{M_{prs}}}$
$\frac{}{(q_m,\sigma_0)\rightarrow(q_0,\sigma\vert_x^{eval^{\sigma_0}(x)})}$, $\frac{}{(q_0,\sigma)\rightarrow(q_1,\sigma'\vert_f^{(b,1)})\Vert(init_b^{I_{S_b}},\sigma_b^{dy}\vert_{b,x_0}^{1,eval^{\sigma}(x)})}$, $\frac{}{(q_1,\sigma')\rightarrow(q_2,\sigma''\vert_g^{(c,1)})\Vert(init_c^{I_{S_c}},\sigma_c^{dy}\vert_{c,x_0}^{1,eval^{\sigma'}_A(x)})}$, $\frac{}{(q_2,\sigma'')\rightarrow(init_a^{I_{S_a}},\sigma_a^{dy}).(q_f,\sigma'')}$, $\frac{\sigma''(f)=(b,\sigma_b'^{dy}(b))}{(q_f,\sigma'')\Vert(ret_b^{I_{S_b}},\sigma_b'^{dy})\rightarrow(q_f,\sigma'')}$, $\frac{\sigma''(g)=(c,\sigma_c'^{dy}(c))}{(q_f,\sigma'')\Vert(ret_c^{I_{S_c}},\sigma_c'^{dy})\rightarrow(q_f,\sigma'')}$, $\frac{}{(ret_a^{I_{S_a}},\sigma_a'^{dy}).(q_f,\sigma'')\rightarrow(q_f,\sigma'')}$	$q_0 \rightarrow (init_b, f) \parallel q_1$, $(ret_b, f) \parallel q_f \rightarrow q_f$, $q_1 \rightarrow (init_c, g) \parallel q_2$, $(ret_c, g) \parallel q_f \rightarrow q_f$, $q_2 \rightarrow init_a.q_f$, $ret_a.q_f \rightarrow q_f$

Abbildung 6.1.: Die (G,G)-PRS-Regelmenge $\rightarrow_{M_{m_{prs}}}$ für den Service M aus Abbildung 4.21. Die Platzhalter $init_x$ und ret_x mit $x \in \{a, b, c\}$ stehen für die jeweiligen benötigten Prozeduren wie in der Nutzungsschnittstelle R_M beschrieben.

Die (G,G)-PRS-basierte Abstraktion $[\![A]\!]_{prs}$ des in Kapitel 4.4 beschriebenen Service A lautet wie folgt: $[\![A]\!]_{prs} = (Z_{A_{prs}}, i_a, \rightarrow_{A_{prs}}, \{r_a\}, \mathcal{D}_{A_{prs}})$ mit:

- $Z_{A_{prs}} = PEX(NODE_A \cup DNODE^{R_A} \cup NODE_A \times VAR \cup (DNODE^{R_A} \times VAR))$,
- $\rightarrow_{A_{prs}}$ in Abbildung 6.2 und

der Platzhalterfunktion für Service A:

$$\mathcal{D}_A(init_a) = (i_a) \tag{6.1}$$

$$\mathcal{D}_A(ret_a) = (r_a) \tag{6.2}$$

Die Regeln $\rightarrow_{A_{prs}}$ sind in Abbildung 6.2 zu sehen. Auch bei der (G,G)-PRS-basierten Service-Abstraktion von Service A ist zu sehen, dass die Regeln mit den zusätzlichen Kellerelementen $(s_a^{I_A}, \sigma_a^{dy})$ und (re_a, σ''') wegfallen, da diese nur zur Übertragung der übergebenen oder zurückgegeben Parameter in die operationale Semantik eingeführt worden.

Die (G,G)-PRS-basierte Abstraktion $[\![B]\!]_{prs} = (Z_{B_{prs}}, i_b, \rightarrow_{B_{prs}}, \{r_b\}, \mathcal{D}_{B_{prs}})$ ist gegebenen durch:

- $Z_{B_{prs}} = PEX(NODE_B \times \{\varphi\})$ mit $\varphi \in VAR$,

Regelmenge des konkreten Verhaltens von Service A, Fig. 4.21	(G,G)-PRS-Regeln $\rightarrow_{A_{prs}}$
$\frac{}{(s_a^{I_A},\sigma_a^{dy})\rightarrow(i_a,\sigma\vert_z^{eval^{\sigma_a^{dy}}(x_0)})}$,	
$\frac{}{(i_a,\sigma)\rightarrow(q_{a1},\sigma'\vert_x^{eval^{\sigma}(z)})}$,	
$\frac{eval^{\sigma'}(x==1)=true}{(q_{a1},\sigma')\rightarrow(q_{a2},\sigma')}$,	
$\frac{eval^{\sigma'}(x==1)=false}{(q_{a1},\sigma')\rightarrow(q_{a4},\sigma')}$,	
$\frac{}{(q_{a2},\sigma')\rightarrow(init_b^{I_{S_b}},\sigma_b^{dy}\vert_{b,a_1}^{1,eval^{\sigma}(x)})\Vert(q_{a3},\sigma''\vert_f^{(b,1)})}$,	$i_a \rightarrow q_{a1}$,
$\frac{}{(q_{a3},\sigma'')\rightarrow(q_{a7},\sigma'')}$,	$q_{a1} \rightarrow q_{a2}$,
$\frac{}{(q_{a4},\sigma')\rightarrow(init_c^{I_{S_c}},\sigma_c^{dy}\vert_{c,x_0}^{1,eval^{\sigma}(x)})\Vert(q_{a_5},\sigma'''\vert_g^{(c,1)})}$,	$q_{a1} \rightarrow q_{a4}$,
$\frac{\sigma'''(g)=(c,\sigma_c^{dy}(c))}{(q_{a5},\sigma''')\Vert(ret_c^{I_{S_c}},\sigma_c'^{dy})\rightarrow(q_{a6},\sigma'')}$,	$q_{a2} \rightarrow (init_b, f) \parallel q_{a3}$,
$\frac{}{(q_{a6},\sigma'')\rightarrow(q_{a7},\sigma'')}$,	$(ret_b, f) \parallel r_a \rightarrow r_a$,
$\frac{eval^{\sigma''}(x==1)=true}{(q_{a7},\sigma'')\rightarrow(q_{a8},\sigma)}$,	$q_{a3} \rightarrow q_{a7}$,
$\frac{eval^{\sigma''}(x==1)=false}{(q_{a7},\sigma'')\rightarrow(q_{a10},\sigma'')}$,	$q_{a4} \rightarrow (init_c, g) \parallel q_{a_5}$,
$\frac{\sigma''(f)=(b,\sigma_b^{dy}(b))}{(q_{a8},\sigma'')\Vert(ret_b^{I_{S_b}},\sigma_b^{dy})\rightarrow(q_{a9},\sigma'')}$,	$(ret_c, g) \parallel r_a \rightarrow r_a$,
$\frac{}{(q_{a9},\sigma'')\rightarrow(re_a,\sigma''')}$,	$(ret_c, g) \parallel q_{a5} \rightarrow q_{a6}$,
$\frac{}{(q_{a10},\sigma'')\rightarrow(q_{a11},\sigma'''\vert_z^{eval^{\sigma}(x)})}$,	$q_{a6} \rightarrow q_{a7}$,
$\frac{}{(q_{a11},\sigma''')\rightarrow(re_a,\sigma''')}$,	$q_{a7} \rightarrow q_{a8}$,
$\frac{\sigma(g)=(c,\sigma_c'^{dy}(c))}{(re_a,\sigma''')\Vert(ret_c^{I_{S_c}},\sigma_c'^{dy})\rightarrow(re_a,\sigma''')}$,	$q_{a7} \rightarrow q_{a10}$,
$\frac{\sigma(f)=(b,\sigma_b'^{dy}(b))}{(re_a,\sigma''')\Vert(ret_b^{I_{S_b}},\sigma_b'^{dy})\rightarrow(re_a,\sigma''')}$,	$q_{a8} \parallel (ret_b, f) \rightarrow q_{a9}$,
$\frac{}{(re_a,\sigma''')\rightarrow(r_a^{I_A},\sigma_b^{dy})}$	$q_{a9} \rightarrow r_a$,
	$q_{a10} \rightarrow q_{a11}$,
	$q_{a11} \rightarrow r_a$

Abbildung 6.2.: Die (G,G)-PRS-Regeln $\rightarrow_{A_{prs}}$ der Zustandsübergänge für den Service A aus Abbildung 4.21.

- $\rightarrow_{B_{prs}}$ in Abbildung 6.3 und

der Platzhalterfunktion für Service B:

$$\mathcal{D}_B(init_b, \varphi) = (i_b, \varphi) \tag{6.3}$$

$$\mathcal{D}_B(ret_b, \varphi) = (r_b, \varphi). \tag{6.4}$$

Die durch Service B zur Verfügung gestellte Prozedur b ist asynchron. Aus diesem Grund führt jeder Ausdruck den Programmpunkt $q \in NODE_B$ und die Information, an welches Future sie gebunden ist mit φ, der den Namen des Futures beinhaltet. Auch bei der Abstraktion von asynchronen Prozeduren fallen die in der konkreten Semantik eingeführten zusätzlichen Programmpunkte weg. Mit der Platzhalterfunktion $\mathcal{D}_B$ können die Platzhalter des bindenden Services aufgelöst werden.

Die (G,G)-PRS-basierte Abstraktion $[\![C]\!]_{prs} = (Z_{C_{prs}}, i_c, \rightarrow_{C_{prs}}, \{r_c\}, \mathcal{D}_{C_{prs}})$ ist gegebenen durch:

Regelmenge der konkreten Verhalten von Service B, Fig. 4.21	(G,G)-PRS-Regeln $\rightarrow_{B_{prs}}$
$\frac{}{(s_b^{I_B},\sigma_b^{dy})\rightarrow(i_b,\sigma\vert_y^{eval^{\sigma_b^{dy}}(x_0)})}$, $\frac{}{(i_b,\sigma)\rightarrow(re_b,\sigma'\vert_y^{eval^{\sigma}(y+1)})}$, $\frac{}{(re_b,\sigma')\rightarrow(r_b^{I_B},\sigma_b'^{dy})}$	$(i_b,\varphi)\rightarrow(r_b,\varphi)$

Abbildung 6.3.: Die (G,G)-PRS-Regeln $\rightarrow_{B_{prs}}$ der Zustandsübergänge für den Service B aus Abbildung 4.21

Regelmenge der konkreten Verhalten von Service C, Fig. 4.21	(G,G)-PRS-Regeln $\rightarrow_{C_{prs}}$
$\frac{}{(s_c^{I_C},\sigma_c^{dy})\rightarrow(i_c,\sigma\vert_{v,c}^{eval^{\sigma_c^{dy}}(x_0),eval^{\sigma_c^{dy}}(c)})}$, $\frac{eval^{\sigma}(v==2)=true}{(i_c,\sigma)\rightarrow(q_{c1},\sigma)}$, $\frac{eval_B^{\sigma}(v==2)=false}{(i_c,\sigma)\rightarrow(q_{c3},\sigma)}$, $\frac{}{(q_{c1},\sigma)\rightarrow(q_{c2},\sigma'\vert_v^{eval^{\sigma}(1)})}$, $\frac{}{(q_{c2},\sigma')\rightarrow(re_c,\sigma'')}$, $\frac{}{(q_{c3},\sigma)\rightarrow(q_{c4},\sigma'\vert_v^{eval^{\sigma}(2)})}$, $\frac{}{(q_{c4},\sigma')\rightarrow(re_c,\sigma'')}$, $\frac{}{(re_c,\sigma'')\rightarrow(r_c^{I_C},\sigma_c'^{dy}\vert_{c,rval}^{eval^{\sigma''}(c),eval^{\sigma''}(v)})}$,	$(i_c,\varphi)\rightarrow(q_{c1},\varphi)$, $(i_c,\varphi)\rightarrow(q_{c3},\varphi)$, $(q_{c1},\varphi)\rightarrow(q_{c2},\varphi)$, $(q_{c2},\varphi)\rightarrow(r_c,\varphi)$, $(q_{c3},\varphi)\rightarrow(q_{c4},\varphi)$, $(q_{c4},\varphi)\rightarrow(r_c,\varphi)$,

Abbildung 6.4.: Die (G,G)-PRS-Regeln $\rightarrow_{C_{prs}}$ der Zustandsübergänge für den Service C aus Abbildung 4.21

- $Z_{C_{prs}} = PEX(NODE_C \times \{\varphi\}))$ mit $\varphi \in VAR$,
- $\rightarrow_{C_{prs}}$ in Abbildung 6.4und

der Platzhalterfunktion für Service C:

$$\mathcal{D}_C(init_c,\varphi) = (i_c,\varphi) \tag{6.5}$$

$$\mathcal{D}_C(ret_c,\varphi) = (r_c,\varphi). \tag{6.6}$$

Die (G,G)-PRS-basierte Abstraktion von Service C unterscheidet sich nicht zu der (G,G)-PRS-basierten Abstraktion von Service B. Beide Services haben keine Nutzungsschnittstellen.

Mit Hilfe der beschriebenen (G,G)-PRS-basierten Service-Abstraktionen von Service M, A, B und C ergibt sich für $SoS[M,A,B,C]$ nach dem Kompositionsverfahren das folgende Zustandsübergangssystem $[\![SoS]\!]_{prs} = (Z_{SoS},\rightarrow_{SoS},q_0,q_f,\mathcal{D}_{SoS})$ mit:

- $Z_{SoS} = Z_{M_{prs}} \cup Z_{A_{prs}} \cup \{(q,f_M),(q,f_A) : (q,\varphi) \in Z_{B_{prs}}\} \cup \{(q,g_M),(q,g_A) : (q,\varphi) \in Z_{C_{prs}}\}$, wobei f_A, f_M das Future f in A bzw. *main* und g_A, g_M das Future g in A bzw. *main* bezeichnet.

- $\rightarrow_{SoS}$ in Abbildung 6.5 und

die Platzhalterfunktion $\mathcal{D}_{SoS}$ für das Service-orientierte System ist definiert durch:

$$\mathcal{D}_{SoS}(init_a) = i_a \tag{6.7}$$

$$\mathcal{D}_{SoS}(ret_a) = r_a \tag{6.8}$$

$$\mathcal{D}_{SoS}(init_b, x) = (i_b, x) \qquad \text{für } x \in \{f_A, f_M\} \tag{6.9}$$

$$\mathcal{D}_{SoS}(ret_b, x) = (r_b, x) \qquad \text{für } x \in \{f_A, f_M\} \tag{6.10}$$

$$\mathcal{D}_{SoS}(init_c, x) = (i_c, x) \qquad \text{für } x \in \{g_A, g_M\} \tag{6.11}$$

$$\mathcal{D}_{SoS}(ret_c, x) = (r_c, x) \qquad \text{für } x \in \{g_A, g_M\} \tag{6.12}$$

Durch das Auflösen der Platzhalter ergibt sich $\rightarrow_{SoS}$ wie in Abbildung 6.5 dargestellt, wobei die Regeln in dieser Abbildung nach Ausgangsservice sortiert sind. Es ist zu sehen, dass für jedes Future die asynchronen Prozeduren b und c kopiert wurden.

Nun wird wie in Abschnitt 4.4.3 das Verhalten bei der Eingabe von $x := 2$ betrachtet, da dies in der konkreten Semantik zu einem Deadlock führt. In Tabelle 6.3 wird in der zweiten Spalte der Lauf der operationalen Semantik aus Abbildung 4.32 gezeigt. Parallel dazu wird in der dritten Spalte der Lauf im (G,G)-PRS-basierten abstrakten Zustandsübergangssystem gezeigt. Die mit $*$ versehenen Einträge in der dritten Spalte symbolisieren den Wegfall der zusätzlichen eingeführten Programmpunkte und damit auch den Wegfall dieser Regel.

Auf den prozess-algebraischen Ausdruck $(q_{a8}, \sigma_{11}).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy})$, der bei Anwendung der $\rightarrow_{SoS_{prs}}$-Regeln entsteht, ist keine Regel mehr Anwendung. Das Zustandsübergangssystem $[\![SoS]\!]_{prs}$ hat damit die Normalform

$$(q_{a8}, \sigma_{11}).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy})$$

erreicht, die nicht dem Finalzustand q_f sondern einen Deadlock d entspricht.

Damit ist gezeigt, dass die (G,G)-PRS-basierte Abstraktion, im Gegensatz zur Petri-Netz-basierten Abstraktion, bei diesem Beispiel den Deadlock konserviert.

6.4. Aussagen über Deadlocks in (G,G)-(PRS)-Abstraktionen

Da die (G,G)-(PRS)-Abstraktionen präziser sind als die Petri-Netz-Abstraktionen sind, liegt es nahe zu vermuten, dass falsche Aussagen über die Abwesenheit von Deadlocks in den (G,G)-(PRS)-Abstraktionen ausgeschlossen sind. Es wird gezeigt, dass dies zwar nicht immer der Fall ist, aber unter bestimmten Voraussetzungen an das Service-

Service M	Service A
$q_0 \to (i_b, f) \parallel q_1,$ $(r_b, f) \parallel q_f \to q_f,$ $q_1 \to (i_c, g) \parallel q_2,$ $(r_c, g) \parallel q_f \to q_f,$ $q_2 \to i_a.q_f,$ $r_a.q_f \to q_f$	$i_a \to q_{a1},$ $q_{a1} \to q_{a2},$ $q_{a1} \to q_{a4},$ $q_{a2} \to (init_b, f) \parallel q_{a3},$ $(ret_b, f) \parallel r_a \to (r_a),$ $q_{a3} \to q_{a7},$ $q_{a4} \to (init_c, g) \parallel (q_{a_5}),$ $(ret_c, g) \parallel r_a \to r_a,$ $(ret_c, g) \parallel (q_{a5}, g) \to q_{a6},$ $q_{a6} \to q_{a7},$ $q_{a7} \to q_{a8},$ $q_{a7} \to q_{a10},$ $(q_{a8}, f) \parallel (ret_b, f) \to q_{a9},$ $q_{a9} \to r_a,$ $q_{a10} \to q_{a11},$ $q_{a11} \to r_a$
Service B	**Service C**
$(i_b, f_M) \to (r_b, f_M),$ $(i_b, f_A) \to (r_b, f_A)$	$(i_c, g_M) \to (q_{c1}, g_M),$ $(i_c, g_M) \to (q_{c3}, g_M),$ $(q_{c1}, g_M) \to (q_{c2}, g_M),$ $(q_{c2}, g_M) \to (r_c, g_M),$ $(q_{c3}, g_M) \to (q_{c4}, g_M),$ $(q_{c4}, g_M) \to (r_c, g_M),$ $(i_c, g_A) \to (q_{c1}, g_A),$ $(i_c, g_A) \to (q_{c3}, g_A),$ $(q_{c1}, g_A) \to (q_{c2}, g_A),$ $(q_{c2}, g_A) \to (r_c, g_A),$ $(q_{c3}, g_A) \to (q_{c4}, g_A),$ $(q_{c4}, g_A) \to (r_c, g_A)$

Abbildung 6.5.: Das Regelsystem $\to_{SoS}$ der abstrakten (G,G)-PRS-Abstraktion des Service-orientierten Systems SoS aus Abbildung 4.26.

Kaktus-keller in Abb. 4.5 $[\![SoS]\!]_{konkr.}$	Lauf in $[\![SoS]\!]_{prs}$	Lauf in $[\![\pi]\!]_{prs}$
(0)	$\underline{(q_m, \sigma_0)} \Rightarrow$	$*$
(1)	$\underline{(q_0, \sigma_1)} \Rightarrow$	$\underline{q_0} \Rightarrow$
(2)	$\underline{(q_1, \sigma_2)} \parallel (s_b, \sigma_1^{dy}) \Rightarrow$	$\underline{q_1} \parallel (i_b, f_M) \Rightarrow$
(3)	$(q_2, \sigma_3) \parallel \underline{(s_b, \sigma_1^{dy})} \parallel (s_c, \sigma_2^{dy}[c := 2, x_0 := 2]) \Rightarrow$	$q_2 \parallel \underline{(i_b, f_M)} \parallel (i_c, g_M) \Rightarrow$
(4)	$(q_2, \sigma_3) \parallel \underline{(i_b, \sigma_4)} \parallel (s_c, \sigma_2^{dy}) \Rightarrow$	$*$
(5)	$(q_2, \sigma_3) \parallel \underline{(re_b, \sigma_5)} \parallel (s_c, \sigma_2^{dy}) \Rightarrow$	$\underline{q_2} \parallel (r_b, f_M) \parallel (i_c, g_M) \Rightarrow$
(6)	$(q_2, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel \underline{(s_c, \sigma_2^{dy})} \Rightarrow$	$*$
(7)	$\underline{(q_2, \sigma_3)} \parallel (r_b, \sigma_3^{dy}) \parallel (i_c, \sigma_6) \Rightarrow$	$*$
(8)	$\underline{(s_a, \sigma_4^{dy})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (i_c, \sigma_6) \Rightarrow$	$\underline{i_a}.q_f \parallel (r_b, f_M) \parallel (i_c, g_M) \Rightarrow$
(9)	$(i_a, \sigma_7).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel \underline{(i_c, \sigma_6)} \Rightarrow$	$*$
(10)	$(i_a, \sigma_7).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel \underline{(q_{c1}, \sigma_6)} \Rightarrow$	$i_a.q_f \parallel (r_b, f_M) \parallel \underline{(q_{c1}, g_M)} \Rightarrow$
(11)	$(i_a, \sigma_7).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel \underline{(q_{c2}, \sigma_8)} \Rightarrow$	$i_a.q_f \parallel (r_b, f_M) \parallel \underline{(q_{c2}, g_M)} \Rightarrow$
(12)	$(i_a, \sigma_7).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel \underline{(re_c, \sigma_8)} \Rightarrow$	$\underline{i_a}.q_f \parallel (r_b, f_M) \parallel (r_c, g_M) \Rightarrow$
(13)	$\underline{(i_a, \sigma_7)}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel \underline{(r_c, \sigma_5^{dy})} \Rightarrow$	$*$
(14)	$\underline{(q_{a1}, \sigma_9)}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$\underline{q_{a1}}.q_f \parallel (r_b, f_M) \parallel (r_c, g_M) \Rightarrow$
(15)	$\underline{(q_{a4}, \sigma_9)}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$\underline{q_{a4}}.q_f \parallel (r_b, f_M) \parallel (r_c, g_M) \Rightarrow$
(16)	$(q_{a5}, \sigma_{10}) \parallel \underline{(s_c, \sigma_6^{dy})}).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$q_{a5} \parallel \underline{(i_c, g_A)}).q_f \parallel (r_b, f_M) \parallel (r_c, g_M) \Rightarrow$
(17)	$(q_{a5}, \sigma_{10}) \parallel \underline{(i_c, \sigma_{12})}).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$*$
(18)	$(q_{a5}, \sigma_{10}) \parallel \underline{(q_{c1}, \sigma_{12})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$q_{a5} \parallel \underline{(q_{c1}, g_A)}.q_f \parallel (r_b, f_M) \parallel (r_c, g_M) \Rightarrow$
(19)	$(q_{a5}, \sigma_{10}) \parallel \underline{(q_{c2}, \sigma_{13})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$q_{a5} \parallel \underline{(q_{c2}, g_A)}.q_f \parallel (r_b, f_M) \parallel (r_c, g_M) \Rightarrow$
(20)	$(q_{a5}, \sigma_{10}) \parallel \underline{(re_c, \sigma_{13})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$*$
(21)	$\underline{(q_{a5}, \sigma_{10})} \parallel \underline{(r_c, \sigma_7^{dy})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$\underline{q_{a5}} \parallel (r_c, g_A).q_f \parallel (r_b, f_M) \parallel (r_c, g_M) \Rightarrow$
(22)	$\underline{(q_{a6}, \sigma_{11})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$\underline{q_{a6}}.q_f \parallel (r_b, f_M) \parallel (r_c, g_M) \Rightarrow$
(23)	$\underline{(q_{a7}, \sigma_{11})}.(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \Rightarrow$	$\underline{q_{a7}}.q_f \parallel (r_b, f_M) \parallel (r_c, g_M) \Rightarrow$
(24)	$(q_{a8}, \sigma_{11}).(q_f, \sigma_3) \parallel (r_b, \sigma_3^{dy}) \parallel (r_c, \sigma_5^{dy}) \not\Rightarrow$ Normalform	$q_{a8}.q_f \parallel (r_b, f_M) \parallel (r_c, g_M) \not\Rightarrow$ Normalform

Tabelle 6.3.: Lauf der konkreten operationalen Semantik im Vergleich zum Lauf in der (G,G)-PRS-basierten Abstraktion des Service-orientierte System aus Abbildung 4.26 bei $x := 2$. $*$ kennzeichnet den Wegfall von Übergängen, die auf zusätzlich eingeführte Programmpunkte zurückzuführen sind.

orientierte System tatsächlich keine falsche Aussagen über Abwesenheit von Deadlocks getroffen werden.

Diese Voraussetzung ist in Definition 6.8 definiert:

Definition 6.8 (Einfach asynchrone Service-orientierte Systeme)**.** Sei π ein Service-orientiertes System und $P = \{p_1, \ldots, p_n\}$ eine Menge verschränkt rekursiver Funktionen. P heißt einfach asynchron, wenn p_i höchstens einen Aufruf von p_j für alle $1 \leq i, j \leq n$ enthält. π heißt einfach asynchron, wenn alle Mengen verschränkt rekursiver Funktionen einfach asynchron sind.

Bemerkung 6.3. *Die Menge P in Definition 6.8 ist eine starke Zusammenhangskomponente im Aufrufgraph von π.*

Das folgende Theorem zeigt, dass für einfach asynchrone Service-orientierte Systeme die PRS-Abstraktion alle Deadlocks erkennt:

Theorem 6.1 (Erhalt von Deadlocks)**.** *Sei π ein einfach asynchrones Service-orientiertes System. Falls die konkrete Semantik $[\![\pi]\!]_{konkret}$ einen Deadlock hat, dann hat auch die (G,G)-PRS-Abstraktion $[\![\pi]\!]_{prs}$ einen Deadlock.*

Der Beweis wird zur besseren Verdeutlichung mithilfe der Kaktuskellerdarstellung geführt. Außerdem wird angenommen, dass falls lokal in jeder aufrufenden Prozedur (synchron oder asynchron) jedes Future an unterschiedliche Prozeduren gebunden wird. Weiterhin heißt ein Teilausdruck $e = (q_1 \parallel \cdots \parallel q_n)$ eines Ausdruckes $(q_1 \parallel \cdots \parallel q_n).q$ *Kette*, wenn $a_1, \cdots, a_n \in Atom$ sind und es keine Atome a' mit $a_1 \parallel \cdots \parallel a_n \parallel a'$ als Teilausdruck von e gibt.

Lemma 1. *Wenn in einem Deadlock d für jede Kette κ von d eine der beiden folgenden Eigenschaften erfüllt ist:*

*(i) für jedes $a_i = (q_i, \beta) \in \kappa$ gilt $q_i :$ **sync** f_i mit einem Future f_i oder*

(ii) die Kette κ hat die Länge 1,

dann ist $d' = \alpha_{prs}(d)$ ein Deadlock in der (G,G)-PRS-basierten Abstraktion $[\![\Pi]\!]_{prs}$.

Lemma 2. *Falls d Deadlock in einer Kette κ ist, dann gilt $\kappa \stackrel{*}{\Rightarrow} d'$, wobei d' ein Deadlock in $[\![\Pi]\!]_{prs}$ ist.*

Beweis 6.1 (Lemma 1)**.** Sei κ eine Kette $(q_1, \sigma_1) \parallel \cdots \parallel (q_n, \sigma_n)$ in Deadlock d. Dann gilt nach (i) $q_i :$ **sync** f_i für $i =, \ldots, n$ und $\alpha_{prs}(\kappa) = (q_1, f_1) \parallel \cdots \parallel (q_n, f_n)$ ist Kette in $\alpha_{prs}(d)$.

$\alpha_{prs}(\kappa)$ ist in Normalform und Teilausdruck von d' in $[\![\pi]\!]_{prs}$, weil keine Regel mit linker Seite $(q_i, f_i) \parallel (q_j, f_j))$ in der sowohl q_i als auch q_j Synchronisationsanweisungen sind.

Sei κ eine Kette der Länge 1. Dann gibt es einen Teilausdruck $e = (e'.(q_2, \sigma_2)) \parallel (q_1, \sigma_1)$ für einen Teilausdruck e'. Wir unterscheiden die zwei Fälle $e' = (q_k, \sigma_k).\cdots.(q_3, \sigma_3)$ für ein $k \geq 3$ und $e' = (q_k \parallel \cdots \parallel q_3)$ für ein $k > 3$.

Fall 1 ($e' = (q_k, \sigma_k).\cdots.(q_3, \sigma_3)$ für ein $k \geq 3$)**.** Dann ist $\alpha_{prs}(e) = q_k.\cdots q_3.q_2 \parallel q_1$ und $\alpha_{prs}(e)$ ist Teilausdruck von d'. Es gibt keine anwendbare Regel auf $\alpha_{prs}(e)$ und $\alpha_{prs}(e)$ ist Teilausdruck von d'

Fall 2 ($e' = (q_k, \sigma_k) \parallel \cdots \parallel (q_3, \sigma_3)$ für ein $k > 3$)**.** Dann handelt es sich bei e' mindestens um eine Kette der Länge 2 und nach (i) ist dies ein Deadlock. Dafür wurde oben schon bewiesen, dass $\alpha_{prs}(e')$ in Normalform ist.

Da d und $d' = \alpha_{prs}(d)$ dieselbe Form haben und alle potentiellen PRS-Regeln (wie Regeln der konkreten Semantik) nur auf Situationen wie in (i) und (ii) anwendbar sind, ist auch d' ein Deadlock.

Beweis 6.2 (Beweis Lemma 2)**.** Nach Lemma 1 genügt es, wenn nur Ketten $\kappa = (q_{r_1,\sigma}) \parallel (q_j, \sigma') \parallel \dots$ mit

- $p_1(\dots)\{\dots q_{r_1} : \mathbf{return};\}$, $\sigma(p_1) = \mu$ für ein $\mu \in \mathbb{N}$,
- $p_2(\dots)\{\mathbf{future}\, f,\, h; \dots q_i : f := p_1(\dots); q_j : \mathbf{sync}\, h; \dots, q_{r_2} : \mathbf{return};\}$ und $\sigma(f) = (p_1, \Phi)$

Dann ist $\kappa' \widehat{=} \alpha_{prs}(\kappa) = (q_{r_1}, f) \parallel (q_j, g) \parallel \dots$ für ein Future g.

Annahme: $\alpha_{prs}(\kappa)$ ist kein Deadlock. Dann muss es in κ' ein Paar (q_k, α) geben mit q_k : **return;** und q_k ist über $h := p_2(\dots)$ aufgerufen worden. Da Futures nur einmal zugewiesen werden können und eindeutig sind, ist der Aufruf in p_2. Also muss p_2 schon vorher aufgerufen worden sein. O.B.d.A. seien die Aufrufe in der konkreten Semantik unmittelbar davor, d.h.

$\kappa = (q_{r_1}, \sigma) \parallel (q_j, \sigma') \parallel (q_{r_3}, \sigma'') \parallel (q_k, \sigma''') \parallel \cdots$, wobei $\sigma''(g) = (p_2, \Phi^{(1)})$, $\sigma'(p_2) = \Phi^{(1)}$ und $\sigma'''(h) = (p_3, \Phi^{(2)})$, $\sigma'''(p_3) = \Phi^{(4)}$.

Dann ist $q_k \neq q_{r_2}$, da sonst κ kein Deadlock wäre. Somit ist q_k : $\mathbf{sync}\, \alpha$ mit $\alpha \neq h$. Damit κ' kein Deadlock ist, muss $\alpha = f$ sein. Somit ist
$\kappa' = (q_{r_1}, f) \parallel (q_j, g) \parallel (q_{r_3}, h) \parallel (q_2, \alpha) \parallel \dots$
$\Rightarrow_{prs} (q_{r_3}, h) \parallel (q_j, g) \parallel (q_{r_1}, f) \parallel (q_2, \alpha) \parallel \dots$
$\Rightarrow_{prs} (q_{j+1}, g) \parallel (q_{r_1}, f) \parallel (q_k, \alpha) \parallel \dots$
$\Rightarrow_{prs} (q_{r_2}, g) \parallel (q_{r_1}, f) \parallel (q_k, \alpha) \parallel \dots$

Da π einfach rekursiv ist, ist $p_1 \neq p_3$. Dann gilt nur:
$(q_{r_2}, g) \parallel (q_{r_1}, f) \parallel (q_k, \alpha) \parallel \dots$
$(q_{r_2}, g) \parallel (q_{k+1}, \alpha) \parallel \dots$
$(q_{r_2}, g) \parallel (q_{r_2}, \alpha) \parallel \dots$
Dies ist nur dann kein Deadlock, falls vorher zwei Mal p_1 aufgerufen wurde, einmal mit Future g und einmal mit Future α. Also ist p_3 ist schon vorher aufgerufen worden und hat $g := p_2(\dots)$ aufgerufen. Dann muss beim Aufrufer p_4 ein $h' := p_3()$; ausgeführt worden sein. Damit ist man in derselben Situation wie am Anfang, mit p_3 als Rolle von p_1 und p_4 als Rolle von p_2.

Da κ' endlich ist, muss es also irgendwann eine Situation in κ' geben, so dass $\kappa' \Rightarrow (q_{r_2}, g) \parallel \dots$ und es gibt kein Aufruf von p_2 aus κ'. Dann gilt immer $\kappa' \overset{*}{\Rightarrow} \kappa''$ und $(q_{r_2}, g) \in \kappa''$, somit gibt es einen Deadlock d mit $\kappa' \overset{*}{\Rightarrow} d$ und $(q_{r_2}, g) \in \kappa''$.

Durch die Lemmata 1 und 2 wurde gezeigt, dass unter den in den Lemmata genannten Voraussetzungen, das Theorem 6.1, dass jeder Deadlock d in der konkreten Semantik $[\![\pi]\!]_{konkret}$ eines Service-orientierten Systems SoS mit der Implementierung π immer zu einem Deadlock d' in der (G,G)-PRS-basierten Abstraktion $[\![\pi]\!]_{prs}$ führt.

Für nicht einfach asynchrone Service-orientierte Systeme können in der (G,G)-(PRS)-Abstraktion jedoch Deadlocks übersehen werden, wie folgendes Beispiel zeigt.

Beispiel 6.1 (Nicht einfach asynchrone Service-orientierte Systeme)**.** Abbildung 6.6 zeigt ein nicht-einfaches Service-orientiertes System, welches in dem in Abb. 6.7 gezeigten Deadlock endet. Die asynchronen Prozeduren a und b werden verschränkt rekursiv aufgerufen, siehe Programmpunkt q_{a1} und q_{b1} und a hat zwei Bindungen eines Aufrufs von b. Somit ist das Service-orientierte System nicht einfach asynchron.

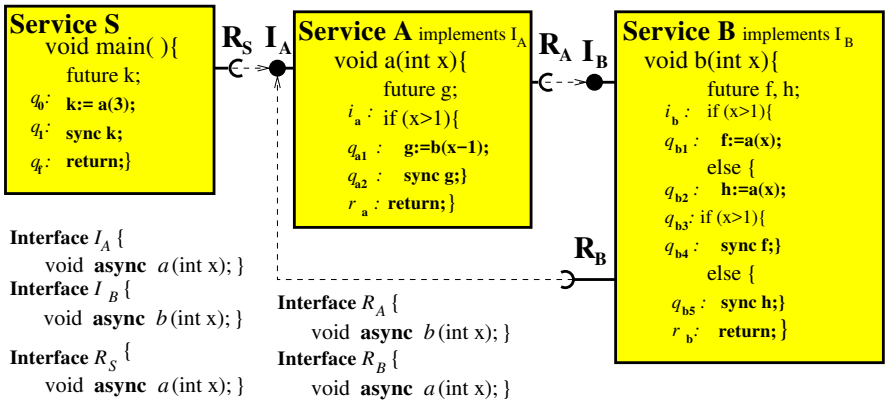


Abbildung 6.6.: (G,G)-PRS-basierte Abstraktion.

Aus der (G,G)-(PRS)-Abstraktion des Deadlocks in Abb. 6.7 ergibt sich folgender Ausdruck:

$$(r_a, b) \parallel (q_{b4}, g) \parallel (q_{a2}, f) \parallel (q_{b5}, g) \parallel (q_{a2}, k) \parallel q_1$$

Dieser Ausdruck kann immer in q_f überführt werden, wie der folgende einzige mögliche Lauf, zeigt (es ist immer die einzige mögliche Anwendungsstelle von Regeln der PRS-Abstraktion unterstrichen):

$(r_a, b) \parallel (q_{b4}, g) \parallel (q_{a2}, f) \parallel (q_{b5}, g) \parallel (q_{a2}, k) \parallel q_1 = \underline{(r_a, b) \parallel (q_{b5}, g)} \parallel (q_{a2}, f) \parallel (q_{b4}, g) \parallel (q_{a2}, k) \parallel q_1$
$\Rightarrow \underline{(r_b, g) \parallel (q_{a2}, f)} \parallel (q_{b4}, g) \parallel (q_{a2}, k) \parallel q_1$
$\Rightarrow \underline{(r_a, f) \parallel (q_{b4}, g)} \parallel (q_{a2}, k) \parallel q_1$
$\Rightarrow \underline{(r_b, g) \parallel (q_{a2}, k)} \parallel q_1 \Rightarrow \underline{(r_a, k) \parallel q_1}$
$\Rightarrow q_f$

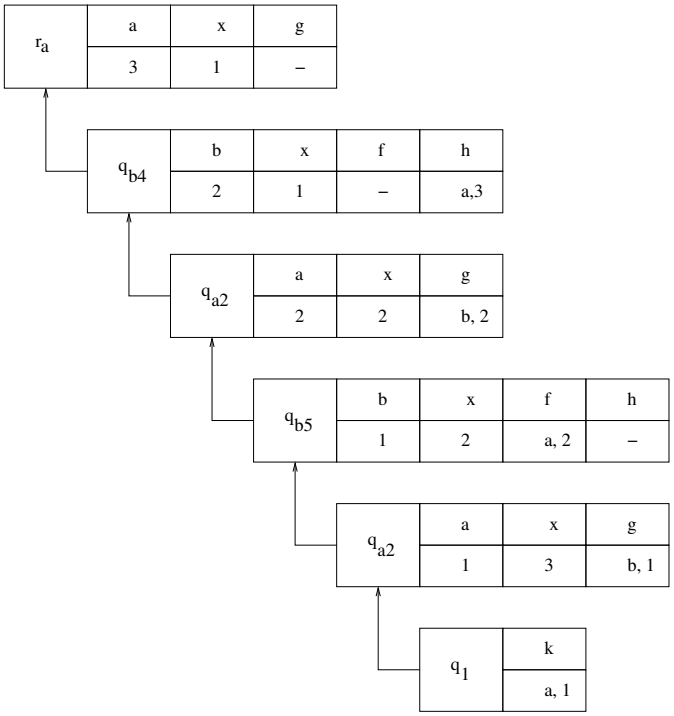


Abbildung 6.7.: Aufrufkette, die bei Ausführung von *main* des Service-orientierten Systems in Abbildung 6.6 entsteht.

Also ist die (G,G)-PRS-basierte Abstraktion frei von Deadlocks

Damit wurde gezeigt, dass für einfache asynchrone Service-orientierte Systeme jeder Deadlock auch in der (G,G)-PRS-basierte Abstraktion erkannt wird. Für nicht einfach asynchrone Service-orientierte Systeme gilt das nicht.

6.5. Zusammenfassung und Diskussion

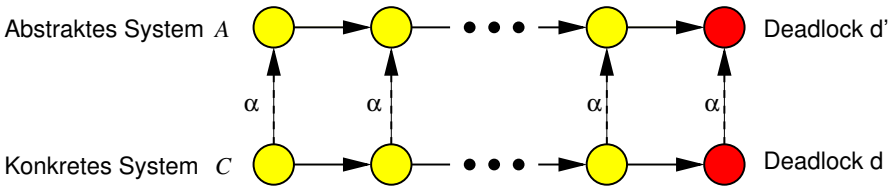


Abbildung 6.8.: (G,G)-PRS-basierte Abstraktion.

In diesem Kapitel wurde von dem Ausführungsmodell der operationalen Semantik, welche logische Kellerelemente plus die Variable, die die Programmpunkte und darüber hinaus den Variableninhalt speichern, dargestellt und mithilfe der Abstraktionsfunktion α_{prs} zu (G,G)-PRS-basierten Abstraktionen abstrahiert. Dabei wurde die Abstraktion erst auf monolithischen Programmen eingeführt, die bei asynchronen Prozeduren auch die Futures ihrer Aufrufe mitführen, so dass nur mit dem richtigen Synchronisationsbefehl synchronisiert werden kann. Darauf aufbauend wurde die Abstraktion von Services erweitert. Im Anschluss wurde der Kompositionsmechanismus für (G,G)-basierte Abstraktionen von Services zu einer (G,G)-basierten Abstraktion eines Service-orientierten Systems definiert. Anhand des in Kapitel 5 präsentierten Service-orientierten System (Abbildung 4.26) wurde die (G,G)-PRS-basierte Abstraktion ausgeführt. Anschließend wurde gezeigt, dass der Lauf, der im Kapitel 5 unter Abbildung 5.3 in der Petri-Netz-basierten Abstraktion nicht zu einen Deadlock, bei der entsprechenden (G,G)-PRS-basierte Abstraktion zu einem Deadlock führt.

Im letzten Abschnitt wird der Zusammenhang zwischen der (G,G)-PRS-basierten Abstraktion und Semantik der Kaktuskeller hergestellt, um darauf aufbauend zu beweisen, dass mithilfe der (G,G)-PRS-basierten Abstraktion für einfach asynchrone Service-orientierte Systeme, jeder Deadlock im konkreten System auch ein Deadlock im abstrakten System ist, (vgl. Abb. 6.8).

Der Grund dafür ist der sogenannte Aufrufkontext. In der (G,G)-PRS-basierten Abstraktion wird dieser über die Kellerelemente gespeichert oder in den Regeln über den „.“-Operator. Die Ausführung in diesem Modell ist damit kontext-sensitiv. Im Vergleich zur Petri-Netz-basierten Abstraktion wird bei mehrmaligen Aufruf einer Prozedur oder Funktion kein neues Petri-Netz erzeugt. Das bereits vorhandene Netz, das den Aufruf einer Prozedur oder Funktion symbolisiert, bekommt eine weitere Marke (vgl. Kapitel 5.3.1). Durch die Kommutativität und Assoziativität des „ $\|$ “-Operators, kann eine entsprechende Synchronisation durchgeführt werden, unabhängig vom Aufrufkontext. Diese Art der Komposition wird kontext-insensitive Komposition genannt, die zu falsch positiven Aussagen in Petri-Netz-basierten Abstraktionen führen kann. Trotzdem kann dies für nicht einfach Service-orientierte Systeme dazu führen, dass die (G,G)-(PRS)-Abstraktion frei von Deadlocks ist, obwohl das konkrete System Deadlocks hat.

Damit lautet das Fazit dieses Kapitels, dass kontext-insensitive Komposition, wie die Komposition bei der Petri-Netz-basierten Abstraktion zu falsch positiven Ergebnissen führen kann. Der kontext-sensitive Ansatz, der (G,G)-PRS-basierten Abstraktion, konserviert alle Deadlocks für einfach asynchrone Service-orientierte Systeme. Zwar ist das Deadlock-Problem für Prozessersetzungssysteme entscheidbar, aber die Prüfung ist aufwendig und uns sind keine Werkzeuge bekannt, die für (G, G)-PRS eine Deadlock-Erkennung prüfen. Daher wird im nächsten Kapitel der Fokus auf die kontext-sensitive und kontext-insensitive Komposition gelegt und diese eingeschränkt, damit Petri-Netz-basierte Ansätze verwendet werden können.

7. Kontext-abhängige Komposition von Service-Abstraktionen

Aus dem letzten Kapitel ist bereits bekannt, dass (G,G)-PRS-basierte Abstraktionen Deadlocks in der konkreten Semantik von einfach asynchronen Service-orientierten Softwaresystemen konservieren können. Somit kann die (G,G)-PRS-basierte Deadlockanalyse für diese Klasse von Service-orientierten Systemen keine falschen positiven Ergebnisse liefern. Im Gegensatz dazu, wurde in Kapitel 5 gezeigt, dass Petri-Netz-basierte Ansätze zu falsch positiven Ergebnissen auch für einfach asynchrone Service-orientierte Systeme führen können. Der Grund für die falsch positiven Ergebnisse ist der Aufrufkontext, der bei Petri-Netz-basierten Ansätzen beziehungsweise bei der Abstraktion verloren geht. Dieses Kapitel untersucht, inwieweit der Kontext betrachtet werden muss, damit alle Deadlocks der (G,G)-(PRS)-basierten Abstraktion erkannt werden.

7.1. Kontext-Insensitivität

In der Petri-Netz-basierten Abstraktion erhält jeder Programmpunkt eine Stelle. Beim Aufruf einer Prozedur oder Funktion erhält die Stelle, die den ersten Programmpunkt im Rumpf der aufgerufenen Methode entspricht und das korrespondierende Petri-Netz dieser Prozedur oder Funktion entspricht, eine Marke. Wird diese Prozedur erneut aufgerufen, so erhält diese Stelle wieder ein Marke. Damit wird vom Kaktus-Keller abstrahiert. Alle Aufrufe einer Prozedur haben eine Transition zum Eintrittspunkt der Prozedur.

In gewisser Weise, ist die Behandlung der Prozeduraufrufe bei der Komposition der Service-Abstraktionen ähnlich wie die Behandlung in kontext-insensitiver interprozeduraler Programmanalysen. Aus diesem Grund wird diese Art der Komposition kontext-insensitive Komposition genannt.

Definition 7.1 (Kontext-Insensitive Komposition)**.** In einer kontext-insensitive Komposition von Service-Abstraktionen kommt jeder Programmpunkt jedes Services nur genau einmal vor.

Ein Beispiel für eine kontext-insensitive Komposition ist die Komposition der Petri-Netz-basierten Abstraktionen in Definition 5.5 in Kapitel 5.

Im Folgenden werden die Auswirkungen einer kontext-insensitiven Komposition diskutiert.

Beispiel 7.1 (Falsch positive Ergebnisse)**.** Das Petri-Netz in Abbildung 5.8 ergibt sich aus der kontext-insensitiven Komposition der Service-Abstraktionen des Service-orientierten Systems in Abbildung 4.26. Im Beweis 5.2 in Kapitel 5 wird gezeigt, dass der in der operationalen Semantik vorkommende Deadlock nicht erkannt wird. Bei der Deadlockanalyse ergeben sich somit falsch positive Ergebnisse. Ein Deadlock wird bei der Analyse nicht erkannt.

Bemerkung 7.1. *[54] zeigt ein weiteres Phänomen als Konsequenz der kontext-insensitiven Komposition: Selbst wenn in der mit der kontext-insensitiven Komposition die Anwesenheit von Deadlocks erkannt wird, so ist jedes Beispiel eines Deadlocks in der Abstraktion unecht. Damit kann durch die Deadlock-Beispiele der Abstraktion nie der echte Deadlock gefunden werden.*

Um diese Auswirkungen zu vermeiden, soll im Folgenden untersucht werden, wie der Aufrufkontext bei der Komposition konserviert werden kann.

7.2. Kontext-Sensitivität und k-kontext-Sensitivität

(G,G)-PRS-basierte Ansätze erhalten im Gegensatz zu Petri-Netz-basierten Ansätzen den Aufrufkontext. Bei der (G,G)-PRS-basierten Abstraktionskomposition wird für jeden Aufruf einer Prozedur oder Funktion ein neuer Kaktuskeller erzeugt, der in Verbindung mit der *Future*-Information den Aufrufkontext speichert. Die (G,G)-PRS-basierte Abstraktionskomposition wird daher als *kontext-sensitive Komposition* bezeichnet.

7.2.1. k-Kontext-Sensitivität

Bei der kontext-sensitiven Komposition werden alle rekursiven Aufrufe aller Prozeduren und Funktionen[1] betrachtet. Kontext-sensitive interprozedurale Programmanalysen sind oft sehr aufwendig, weshalb oft die Rekursionstiefe auf eine Konstante $k \in \mathbb{N}$ beschränkt wird und alle Aufrufe größere Rekursionstiefe kontext-insensitiv in einen zusammengefasst werden. Dieser Unterabschnitt überträgt diese auf die Abstraktionen:

Definition 7.2 (k-kontext-sensitive Abstraktion)**.** Sei $[\![\Pi]\!]_{prs} = (Z_{prs}, \rightarrow_{prs}, q_{0_{prs}}, q_{f_{prs}})$ die kontext-sensitive (G,G)-PRS-basierte Abstraktion eines Service-orientierten Systems

[1] In diesem Kapitel umfasst der Begriff Prozedur auch den Begriff der Funktion.

	Regel aus $\rightarrow_k$	Regel aus $\rightarrow_{prs}$
(1)	$(q, \gamma_k) \rightarrow_k (q', \gamma_k)$	falls $q \rightarrow q'$
(2)	$(q, \gamma_k) \rightarrow_k (q', \gamma_k) \parallel (q'', \gamma_k)$	falls $q \rightarrow q' \parallel q''$
(3)	$((q, f), \gamma_k) \parallel (q', \gamma'_k) \rightarrow_k (q'', \gamma'_k)$	falls $(q, f) \parallel q' \rightarrow (q''), q : r_f, q' : ret$ oder $q' : sync\, f$;
(4)	$(q, \gamma_k) \rightarrow_k (q', \gamma_k\vert_p^{\gamma_k(p)-1}).(q'', \gamma_k)$	falls $q : p()$; und $q \rightarrow q'.q'', \gamma_k(p) < k$
(5)	$(q, \gamma_k) \rightarrow_k (q', \gamma_k)$	falls $q : p()$; und $q \rightarrow q'.q'', \gamma_k(p) = k$
(6)	$(q, \gamma'_k).(q', \gamma_k) \rightarrow_k (q'', \gamma_k)$	falls $q.q' \rightarrow q''$ und $\gamma(proc(q')) \leq k$
(7)	$(q, \gamma_k) \rightarrow_k (q'', \gamma_k)$	falls $q.q' \rightarrow q''$ und $\gamma_k(proc(q)) = k$

Tabelle 7.1.: Die Regeln $\rightarrow_k$ der k-kontext-sensitiven Abstraktion $[\![\Pi]\!]^k_{prs}$.

mit der Implementierung Π gemäß Def. 6.3, $PROC$ die Menge der synchronen Prozeduren von Π, $k \in \mathbb{N}$ und $proc(q)$ die Prozedur, zu der $q \in NODE$ gehört. Das PRS Zustandsübergangssystem $[\![\Pi]\!]^k_{prs} = (Z_k, \rightarrow_k, q_k^{(k)}, q_f^{(k)})$ mit:

- $Z_k = Z \times \{0, \ldots, k\}^{proc}$,
- $\rightarrow_k$ siehe Tabelle 7.1,
- $q_0^{(k)} = (q_0, \mathbf{0})$, wobei $\mathbf{0}(p) = 0$ für alle $p \in PROC$ und
- $q_j^{(k)} = (q_f, \mathbf{0})$,

heißt k-kontext-sensitive Abstraktion von Π.

Bemerkung 7.2. *Bei Rekursionstiefe k wird der Keller nicht mehr erweitert. Also bei Erreichen von Rekursionstiefe k kann das entsprechende Kellerelement abgebaut werden, muss aber nicht, da die tatsächliche Rekursionstiefe > k unbekannt ist. Daher ist sowohl die Regel* (6) *als auch die Regel* (7) *aus Tabelle 7.1 anwendbar.*

Bemerkung 7.3. *Die* 0*-kontext-sensitive Abstraktion sind die Petri-Netz Abstraktionen gemäß Definitionen 5.1, 5.2 und 5.3. Dabei handelt es sich also um kontext-insensitive Kompositionen.*

Bemerkung 7.4. *Die k-kontext-sensitive Komposition von Service-Abstraktionen sind analog den Def. 6.6 und 6.7 definiert.*

Beispiel 7.2 (1-kontext-sensitive Komposition)**.** Das Beispiel 5.2 in Kapitel 5 zeigt ein service-orientiertes System, bei dem die kontext-insensitive Komposition dazu führt, dass Deadlocks nicht erkannt werden. Abb. 7.1 stellt die kontext-insensitive Komposition der 1-kontext-sensitiven Komposition gegenüber. Mit der 1-kontext-sensitiven Komposition kann jeder Aufruf einer Prozedur als Petri-Netz expandieren. Damit wird die 1-kontext-sensitive Komposition auf der rechten Seite erhalten. Der Unterschied besteht darin, dass für jeden Aufruf der Services B und C eine Kopie entsteht. In der (G,G)-PRS-basierten Abstraktion wird dies dadurch erreicht, dass statt der Programmpunkte Paare bestehend aus diesen Programmpunkten und Futures der Aufrufer betrachtet werden.

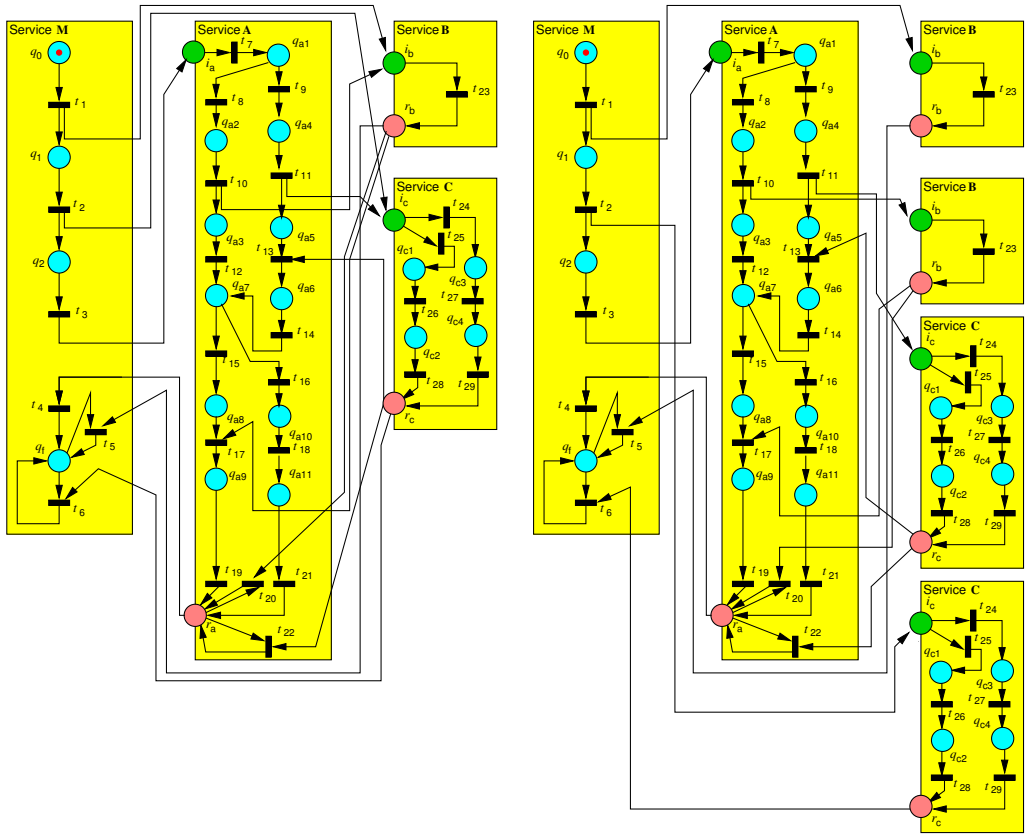


Abbildung 7.1.: Kontext-insensitive Komposition (links) im Vergleich zur 1-kontext-sensitiven Komposition (rechts) der Petri-Netz-basierten Abstraktionen aus Kapitel 5.3.1.

Das Service-orientierte System aus Beispiel 5.2 ist einfach asynchron, da es keine rekursiven Aufrufe enthält – weder bei synchronen noch bei asynchronen Prozeduren. Deshalb erkennt die kontext-sensitive Komposition den Deadlock. Da jeder Prozedur nur nicht-rekursiv aufgerufen wird, erkennt auch die 1-kontext-sensitive Komposition den Deadlock.

7.2.2. 1-Kontext-Sensitivität

Theorem 7.1 (1-Kontext-sensitive Komposition)**.** *Sei* $[\![\Pi]\!]_{prs} = (Z, \rightarrow, q_0, q_f)$ *die kontext-sensitive Abstraktion eines Service-orientierten Systems mit der Implementierung* Π *und* $[\![\Pi]\!]^1_{prs} = (Z_1, \rightarrow_1, q_0^{(1)}, q_f^{(1)})$ *die 1-kontext-sensitive Abstraktion von* Π*. Wenn* $[\![\Pi]\!]_{prs}$ *einen Deadlock hat, dann hat auch* $[\![\Pi]\!]^1_{prs}$ *einen Deadlock.*

Bevor mit dem Beweis begonnen wird, soll kurz eine Beweisskizze gegeben werden: Wenn $\alpha_{prs}(d)$ ein Deadlock in der kontext-sensitiven Komposition der (G,G)-PRS-basierten Abstraktion des Service-orientierten Systems ist, dann gibt es ein oberstes Astelement (zu einer Prozedur a gehörend) deren Ausführung in einem Unterdeadlock d' von $\alpha_{prs}(d)$ endet. Es gibt also einen Lauf vom Startpunkt i_a von Prozedur a zum Deadlock d'. $i_a \Rightarrow d'$ ist auch möglicher Lauf in a. Und damit ist d' ein Deadlock im Petri-Netz, dass durch 1-kontext-sensitive Komposition entstanden ist.

Zum Beweis von Theorem 7.1 werden folgende Definitionen benötigt:

Definition 7.3 (Rekursionstiefe von p)**.** Die Rekursionstiefe von p ist eine Funktion $rek_p : PEX(Z) \to \mathbb{N}$ mit:
$rek_p(a) \widehat{=} \begin{cases} 1, & \text{falls } proc(a) = p \\ 0, & \text{sonst} \end{cases}$ für alle $a \in Z \cup \{\epsilon\}$
$rek_p(e_1, e_2) = rek_p(e_1) + rek_p(e.2)$ und
$rek_p(e_1 \parallel e_2) = \max(rek_p(e_1), rek_p(e_2))$.

Definition 7.4 (Anzahl der Aufrufe von p)**.** Die Anzahl anz der Aufrufe von p ist eine Funktion $anz : PEX(Z) \to \mathbb{N}$ mit:
$anz_p(a) \widehat{=} \begin{cases} 1, & \text{falls } proc(a) = p \\ 0, & \text{sonst} \end{cases}$ für alle $a \in Z$.
$anz_p(e_1, e_2) = anz_p(e_1) + anz_p(e.2)$ und
$anz_p(e_1 \parallel e_2) = anz_p(e_1) + anz_p(e_2)$.

Definition 7.5 (Projektion auf Aufruf einer Prozedur p mit größter Rekursionstiefe)**.** Die Projektion auf Aufruf einer Prozedur p mit größter Rekursionstiefe sei eine Funktion $last_p : PEX(Z) \to PEX(Z)$ mit:
$last_p(e) = e$, falls $rek_p(e) \leq 1$
$last_p(e_1 \parallel e_2) \widehat{=} \begin{cases} last_p(e_1) \parallel, & \text{falls } rek_p(e_1) = rek_p(e_1 \parallel e_2) \\ e_1 \parallel last_p(e_2), & \text{sonst} \end{cases}$.
$last_p(e_1.e_2) = last_p(e_1).e_2$ und
$last_p(q) \widehat{=} \begin{cases} i_p, & \text{falls } proc(q) = p \\ \epsilon, & \text{sonst} \end{cases}$.

Definition 7.6 (Kontext)**.** Ein Kontext K ist ein prozess-algebraischer Ausdruck wobei $K \in PEX(Z \uplus \{\bullet\})$, so dass $anz_\bullet(K) = 1$ ($anz_\bullet$ ist analog anz_p in Def. 7.4 definiert), d.h. K ist induktiv wie folgt definiert:

- $K = \bullet$
- $K = K_1 \parallel e_2$ für einen Kontext K_1 und $e_2 \in PEX(Z)$
- $K = e_1 \parallel K_2$ für einen Kontext K_2 und $e_1 \in PEX(Z)$
- $K = K_1.e_2$ für einen Kontext $K_1, e_1 \in PEX(Z)$

Die Anwendung von K auf ein $e \in PEX(Z)$ ist wie folgt definiert:

- $K[e] = e$, falls $K = \bullet$
- $K[e] = K_1[e] \parallel e_2$, falls $K = K_1 \parallel e_2$
- $K[e] = e_1 \parallel K_2[e]$, falls $K = e_1 \parallel K_2$
- $K[e] = K_1[e].e_2$, falls $K = K_1.e_2$

Ein Kontext K heißt in Normalform, wenn $K[d]$ in Normalform für alle Normalformen d.

Lemma 3. *Falls $[\![\Pi]\!]_{prs}$ einen Deadlock d mit $rek(d) \leq 1$ für alle $p \in PROC$ besitzt, dann gibt es einen Deadlock d' in $[\![\Pi]\!]^1_{prs}$, so dass $d' = \chi(d)$, wobei $\chi : Z_1 \to Z$ mit:*

- $\chi((a, \gamma)) = a$
- $\chi(e_1 \parallel e_2) = \chi(e_1) \parallel \chi(e_2)$
- $\chi(e_1.e_2) = \chi(e_1).\chi(e_2)$

Lemma 4. *Sei $q_0 \stackrel{*}{\Rightarrow} d$ wobei d ein Deadlock mit $rek_p(d) > 1$ ist. Dann gibt es einen Deadlock d' mit $q_0 \stackrel{*}{\Rightarrow} d'$, $anz_p(d') < anz_p(d)$ und $anz_{p'}(d') \leq anz_{p'}(d)$ für alle $p' \in PROC$.*

Beweis 7.1 (Theorem 7.1)**.** Durch Induktion mit Lemma 4 gibt es einen Deadlock d mit $q_0 \stackrel{*}{\Rightarrow} d$ und $rek_p(d) \leq 1$ für alle $p \in PROC$. Nach Lemma 3 gilt dann (wiederum über die Induktion der Länge) $q_o^{(1)} \stackrel{*}{\Rightarrow} d'$ mit $\chi(d') = d$ und d' ist in Normalform.

Beweis 7.2 (Lemma 3)**.** Sei $q_0 \Rightarrow q_1 \Rightarrow \cdots \Rightarrow q_n = d$ ein Lauf in $[\![\Pi]\!]_{prs}$, dann gibt es einen Lauf $q_0^{(1)} \Rightarrow \cdots \Rightarrow q_n^{(1)} = d'$ mit $\chi(d') = d$. Es wird gezeigt, dass wenn d' kein Deadlock ist, dann ist d auch kein Deadlock.

Wenn d' kein Deadlock ist, dann gibt es ein $d'' \in PEX(Z_1)$, so dass $d' \Rightarrow_1 d''$. Sei $\langle h_1 \Rightarrow_1 g_1, \ldots, h_n \Rightarrow_1 g_n \rangle$ eine Herleitung von $d' \Rightarrow_1 d''$, d.h. $h_n = d'$, $g_n = d''$. Dann gibt es eine Herleitung $\langle h'_1 \Rightarrow g'_1, \ldots, h'_n \Rightarrow g'_n \rangle$ und $h'_i = \chi(h)$, d.h. es gibt ein g'_n mit $d \Rightarrow g'_n$.

Induktionsanfang: $h_1 \Rightarrow_1 g_1$: Somit ist $h_1 \to_1 g_1$. Nach Konstruktion von $\to_1$ ist $\chi(h_1) \to g'_1$ für ein $g' \in PEX(Z)$, daraus folgt $h'_1 \to g'_1$

Induktionsschritt: Es sei $h_k \Rightarrow_1 g_k$ hergeleitet worden. Dann ist (S), (P_1) oder (P_2) angewendet worden. Wir zeigen den Fall (S). Die Fälle (P_1) und (P_2) werden analog bewiesen.

Da (S) angewendet wurde, gibt es ein s, h_j und g_j mit $h_k = h_j.s$ und $g_k = g_j.s$ und entsprechend dazu $h_j \Rightarrow_1 g_j$, für $j < k$. Nach IH gibt es ein $h'_j = \chi(h_j)$ und g'_j mit $h'_j \Rightarrow g'_j$. Somit ist $\chi(h_k) = \chi(h_j).\chi(s) = h'_j.s' \Rightarrow g'_j.s'$ mit $s' = \chi(s)$.

Bemerkung 7.5. *$rek_p(q_j) > 1$ ist möglich.*

Vor dem Beweis von Lemma 4 werden weitere Lemmata benötigt.

Lemma 5. *Es gilt:*

(a) $anz_p(K[e]) \leq anz_p(K[e'])$, falls $anz_p(e) \leq anz_p(e')$,

(b) $anz_p(K[e]) < anz_p(K[e'])$, falls $anz_p(e) \leq anz_p(e')$,

(c) falls $e \stackrel{*}{\Rightarrow} e'$ gilt auch $K[e] \stackrel{*}{\Rightarrow} K[e']$ und umgekehrt.

Beweis 7.3 (Lemma 5)**.** (a) und (b) folgen direkt aus der Def. 7.4 von anz_p und der Def. 7.6 von $K[e]$ über die Induktion der Def. von $K[e]$.

(c) wird bewiesen über die Konstruktion einer Herleitung über den Aufbau von $K[e]$.

Induktionsanfang: Falls $K[e] = e$, ist $K = \bullet$ und $K[e'] = e'$, $K[e] \Rightarrow K[e']$ folgt das für $K = \bullet$ direkt aus der Voraussetzung.

Induktionsschritt: Ist $K = K_1 \parallel e_1$ dann gilt nach IH: $K_1[e] \Rightarrow K_1[e']$. Mit (P_1) ergibt sich dann $K[e] \Rightarrow K[e']$.
Die Fälle $K = e_1 \parallel K_2$ und $K = K_1 \parallel e_2$ werden analog bewiesen.

Die Umkehrung kann analog bewiesen werden, nur dass zunächst gezeigt wird, dass bei der Anwendung einer PRS-Regel aus $K[e] \stackrel{1}{\Rightarrow} K[e']$ auch $e \stackrel{1}{\Rightarrow} e'$ folgt. Durch Induktion über die Länge des Laufs von $K[e] \Rightarrow K[e']$ folgt dann die Umkehrung.

Lemma 6. *Sei d ein Deadlock mit Rekursionstiefe $rek_p(d) > 1$. Dann gilt $i_p \Rightarrow d'$ für eine Normalform d' mit $rek_p(d') = 1$.*

Beweis 7.4 (Lemma 6)**.** Nach Def. 7.5 von $last_p$ ist $last_p(d) = K[i_p]$ für eine Normalform, Kontext K. Da $last_p(d) \Rightarrow d$ und K in Normalform, muss es eine Normalform d' geben mit $K[d'] = d$. Somit gilt $i_p \Rightarrow d'$ und somit $anz(d') = 1$. Nach Def. 7.5 von $last_p$ und Def. 7.4 von anz_p gilt $anz_p(K[i_p]) = anz_p(d)$.

Beweis 7.5 (Lemma 4)**.** Ein Deadlock d kann immer über einen Lauf $q_0 \overset{*}{\Rightarrow} K_1[q] \overset{1}{\Rightarrow} K_1[i_p.q'] \overset{*}{\Rightarrow} K_1[e.q'] = d$, wobei K_1 in Normalform und $anz_p(K_1[a]) = 0$ ist, hergeleitet werden. Da $rek_p(d) > 1$ ist, muss also $anz_p(e) > 1$ gelten. Somit existiert nach Lemma 6 eine Normalform d' mit $i_p \Rightarrow d'$ und $anz_p(d') = 1 < anz_p(e)$. Dann gibt es auch einen Lauf $q_0 \overset{*}{\Rightarrow} K_1[q] \overset{1}{\Rightarrow} K_1[i_p.q'] \overset{*}{\Rightarrow} K_1[d'.q']$.

Da $anz_p(K_1[q]) = 0$, $anz_p(i_p.q') = 1 + anz_p(q')$ und $anz(d') = 1$ gilt $anz_p(K_1[d', q']) = 1 < anz_p(d)$. Für alle anderen Prozeduren gilt $anz_p(d'.q) \leq anz_p(e.q)$ und somit gilt $anz_p(K_1[d', q']) \leq anz_p(d)$. Da d' Normalform ist, ist auch $d'.q$ eine Normalform. Weil K_1 in einer Normalform ist, ist nach Def. 7.6 auch $K_1[d'.q]$ in Normalform.

7.3. Zusammenfassung und Diskussion

Kontext-insensitive Kompositionen können auch bei einfach asynchronen Service-orientierten Systemen zu falschen Abwesenheitsaussagen von Deadlocks führen, den sogenannten falsch positiven Ergebnissen. Ein Deadlock in der konkreten Semantik eines Service-orientieren Systems muss nicht unbedingt ein Deadlock in der abstrakten Semantik sein. Außerdem können kontext-insensitive Kompositionen zu nicht-korrigierbaren unechten Gegenbeispielen führen, d.h. jeder Deadlock im abstrakten System mit kontext-insensitiver Komposition ist unecht, obwohl das konkrete System einen Deadlock enthält.

Bei kontext-sensitiver Komposition kann der Aufrufkontext und damit Deadlocks für einfache asynchrone Service-orientierte Systeme erhalten bleiben. Neben der Unterscheidung zwischen kontext-insensitiver und kontext-sensitiver Komposition, kann bei der kontext-sensitiven Komposition zwischen k-kontext-sensitiver und 1-kontext-sensitiver Komposition im abstrakten System unterschieden werden. Falls das konkrete einfach asynchrone Service-orientierte System einen Deadlock hat, so hat auch das durch die 1-kontext-sensitive Komposition entstandene abstrakte System einen Deadlock. Damit wird für einfach asynchrone Service-orientierte Systeme bei 1-kontext-sensitiver Komposition die Anwesenheit von Deadlocks erkannt und somit kann ähnlich wie in Beispiel 7.2 durch Expandieren jedes Aufrufs in sein Petri-Netz eine 1-kontext-sensitive Petri-Netz-Abstraktion erstellt werden. Damit wird der Einsatz von Petri-Netz-Werkzeugen ermöglicht.

8. Zusammenfassung und Ausblick

8.1. Zusammenfassung

Die Hauptresultate dieser Arbeit sind:

1. Reguläre Petri-Netz-basierte Ansätze mit kontext-insensitiven Kompositionsmechanismus können zu falsch positiven Ergebnissen führen. Ein Deadlock im realen Service-orientierten System wird nicht erkannt und wird als deadlockfrei klassifiziert.

2. Deadlockanalyseverfahren für einfach asynchrone Service-orientierte Systeme führen auf Basis von (G,G)-PRS-basierten Abstraktionen nicht zu falsch positiven Ergebnissen.

3. Falsch positive Ergebnisse bei Petri-Netz-basierten Aufrufen sind zurückzuführen auf den fehlenden Aufrufkontext.

4. Der Aufrufkontext kann auf die Tiefe 1 reduziert werden und Deadlocks bleiben erhalten.

5. Der Aufrufkontext kann bei Petri-Netzen durch eine kontext-sensitive Komposition erhalten bleiben und durch das Resultat 4. reicht ein Auffalten bis zur Rekursionstiefe 1 aus, um Deadlocks zu erhalten.

Bemerkung 8.1. *Die falsch positiven Ergebnisse sind im Punkt 1. auch für nicht einfach asynchrone Service-orientierte Systeme möglich.*

Ziel der Arbeit war es, unter den genannten Anforderungen, wie zum Beispiel unbeschränkte Nebenläufigkeit und Rekursion, folgende Fragestellungen zu untersuchen:

- Führen bisherige Ansätze zur Deadlockanalyse zu falsch positiven Ergebnissen?

- Falls bisherige Verfahren zu falsch positiven Ergebnissen führen, existieren alternative Ansätze?

- Ist die Umsetzung des vorgeschlagenen Ansatzes zur Deadlockanalyse prinzipiell möglich?

In Kapitel 4 wird eine Beispielsprache sowie eine dazugehörige Kaktuskeller-Semantik definiert.

Auf der Grundlage der Beispielsprache und der Kaktuskeller-Semantik werden Petri-Netz-basierte Deadlockanalyseverfahren in Kapitel 5 untersucht. Es wird gezeigt, dass bisherige Verfahren zur Deadlockanalyse für Service-orientierte Systeme zu falsch positiven Ergebnissen führen können (Theorem 5.2). Im realen System ist ein Deadlock vorhanden (Kap. 4), dieser wird durch die Deadlockanalyse nicht erkannt und das System wird als deadlockfrei klassifiziert. Der Grund dafür ist der Verlust des Aufrufkontext bei der Service-Abstraktion und der anschließenden Komposition. Bei der Abstraktion eines konkreten Systems wird keine Obermenge über alle möglichen Läufe eines konkreten Systems gebildet, die ausreichend für das Erreichbarkeitsproblem und andere Sicherheitsbedingungen ist.

Im weiteren Verlauf, in Kapitel 6 wurde gezeigt, dass eine Deadlockanalyse auf Grundlage der höchsten PRS-Stufe durchgeführt werden kann, den (G,G)-PRS [40]. Im Bezug auf den Zusammenhang zwischen der (G,G)-PRS-basierten Abstraktion und der Kaktuskeller-Semantik wird hier gezeigt, dass die (G,G)-PRS-basierte Abstraktion für einfach asynchrone Service-orientierte Systeme, jeder Deadlock im konkreten System in der Abstraktion konserviert werden kann. Der Grund für die Konservierung ist der Aufrufkontext, der als Kellerelement im Kaktuskeller gespeichert wird. Da hier der Aufrufkontext gespeichert wird, heißt dieser Ansatz kontext-sensitiv. Im Gegensatz dazu, sind Petri-Netz-basierte Verfahren kontext-insensitiv, da sie den Aufrufkontext nicht speichern.

Zwar ist das Deadlock-Problem für (G,G)-PRS entscheidbar, aber die Prüfung ist aufwendig und uns sind keine Werkzeuge bekannt. Daher wird in der Arbeit im Kapitel 7 die Kontextsensitivität bzw. die kontext-sensitive Komposition betrachtet. Um den Aufrufkontext auch in Petri-Netzen konservieren zu können, wird eine kontext-sensitive Komposition der Service-Abstraktionen definiert, die den Aufrufkontext konserviert. Neben der Unterscheidung der kontext-insensitiven Komposition und der kontext-sensitiven Komposition, ist es außerdem möglich, die kontext-sensitive Komposition zu unterscheiden zwischen k-kontext-sensitiver Komposition und 1-kontext-sensitiver Komposition. Außerdem konnte bewiesen werden, dass wenn ein Deadlock in einem einfach asynchronen Service-orientierten Softwaresystem besteht, so hat auch die 1-kontext-sensitive Abstraktion dieses Systems einen Deadlock, womit der Einsatz Petri-Netz-basierter Werkzeuge wieder möglich ist.

Daher ergeben sich auf die in der Einleitung offenen Fragestellung, folgende Antworten:

Führen bisherige Ansätze zur Deadlockanalyse zu falsch positiven Ergebnissen?

Ja, bisherige Ansätze zur Deadlockanalyse können zu falsch positiven Ergebnissen führen (s. Kap. 5).

Falls bisherige Verfahren zu falsch positiven Ergebnissen führen, existieren alternative Ansätze?
Ja, es existieren alternative Ansätze. Zum einen (G,G)-PRS-basierte Ansätze wie in Kapitel 6 und zum anderen sind auch Petri-Netz-basierte Verfahren einzusetzen, jedoch muss die Komposition eine 1-kontext-sensitive Komposition sein. Ausnahmen sind bei beiden Verfahren nicht einfach asynchrone Service-orientierte Systeme (Def. 6.8 einfach asynchrone Service-orientierte Systeme).

Ist die Umsetzung des vorgeschlagenen Ansatzes zur Deadlockanalyse prinzipiell möglich?
Ja, da das Erreichbarkeitsproblem für Petri-Netz und Prozessersetzungssysteme entscheidbar ist [39].

Fazit

Der in der Arbeit vorgestellte Ansatz zur Deadlockanalyse von Service-orientierten Softwaresystemen genügen folgenden Randbedingungen:

1. Es handelt sich um einen abstraktions-basierten Ansatz, der es ermöglicht bereits implementierte Services auf Deadlocks zu untersuchen. Dabei werden auf Techniken des Übersetzerbaus benutzt [3]. Jeder Kontrollstruktur unserer Beispielsprache wird eine Abstraktion vorgegeben. Das Erreichbarkeitsproblem für Petri-Netze und PRS ist entscheidbar [39]

2. Der vorgeschlagene Black-Box-Paradigma bleibt erhalten. Das heißt, dass eine Deadlockanalyse auch ohne den Quellcode der Services durchführbar ist, indem ähnlich zu [11] die Abstraktionen, zum Beispiel über die Schnittstellenbeschreibungen, veröffentlicht werden. Damit wird lediglich der Kontrollfluss eines Services veröffentlicht, nicht aber die Implementierung.

3. Unbeschränktes paralleles, unbeschränktes rekursives Verhalten sowie rekursive Rückrufe zwischen den Services oder Komponenten werden durch den Kontext in Petri-Netz-basierten Ansätzen und durch die entsprechenden Kellerelemente konserviert.

Eine Klassifikation mit den in Kapitel 2 vorgestellten verwandten Arbeiten wird in Tabelle 8.1 zusammenfassend dargestellt. Dabei hat die Symbolik folgende Bedeutung: '++' die Eigenschaft wird erfüllt, '+' die Eigenschaft teilweise erfüllt, '−' die Eigenschaft

wird teilweise nicht erfüllt, '−−' die Eigenschaft wird nicht erfüllt und der Eintrag '0' bedeutet, dass es unbekannt ist, ob diese Eigenschaft erfüllt wird oder nicht. Der Punkt *Model Checking Ansatz* ist im Sinne von es können keine falsch positiven Ergebnisse auftreten.

Bei Betrachtung der Tabelle 8.1 wird deutlich, dass der überwiegende Anteil der Arbeiten keine Rekursion oder nur beschränkte Rekursion modellieren kann. Das ist auch der Grund für die Kennzeichnung der Spalte *Model-Checking Ansatz* mit dem Eintrag '0'. Das Auftreten von falsch positiven Ergebnissen unter den Voraussetzungen dieser Arbeit, kann nicht garantiert werden. In einigen Arbeiten wird außerdem bereits darauf hingewiesen, dass es zu falsch positiven Aussagen kommen kann [23].

Bemerkung 8.2. *Durch die Einschränkung auf einfach asynchrone Service-orientierte Systeme (s. Hauptresultate, Pkt. 2) wird die Spalte unbeschränkte Nebenläufigkeit mit einem '+' gekennzeichnet.*

8.2. Ausblick

Im letzten Abschnitt werden wir weitere wissenschaftliche Fragestellungen, die sich aus dieser Forschungsarbeit ergeben haben, nennen und mögliche Erweiterungen oder Verbesserungen vorstellen.

Eine wissenschaftliche Fragestellung ergibt sich aus der Arbeit von [29]. Hier wird die Ausnahmebehandlung und deren Auswirkungen auf das Verfahren zur Protokollkonformitätsprüfung untersucht. Das Verhalten von Ausnahmebehandlungen kann von den (G,G)-PRS modelliert werden. Die Auswirkungen von Ausnahmebehandlungen auf die Deadlockanalyse sollte ebenfalls betrachtet werden.

Der Ansatz ist konservativ. Im abstrakten Modell kann es eine Vielzahl an möglichen Läufen geben, die im realen Service-orientierten System nicht auftreten, vgl. dazu mit dem Kapitel 6. Daher stellt sich die Frage, ob durch eine Datenflussanalyse, die durch Überapproximation entstandenen falsch negativen Ergebnisse, ausgeschlossen werden können.

In diesem Zusammenhang stellt sich außerdem die Frage, ob es einen Ansatz gibt, der es ermöglicht, unechte falsche Gegenbeispiele vorher zu identifizieren oder deren Auftreten vorherzusagen [54].

Außerdem bleibt zu untersuchen, wie groß die 1-kontext-sensitiven Abstraktionen werden und ob diese dann noch praktikabel sind. Dazu müssen eine Reihe von Fallstudien durchgeführt werden und der Ansatz mit Petri-Netz Werkzeugen implementiert werden.

		unbeschränkte Rekursion	unbeschränkt Nebenläufigkeit	Abstraktions-basierter Ansatz	Berücksichtigung des Kontextes bei Komposition	Model Checking Ansatz	Service-orientierte Systeme vs. monolithischen Systemen
1.	[2]	−−	+	−−	0	0	++
2.	[53]	−−	++	++	0	0	++
3.	[51]	−−	+	++	0	0	−−
4.	[18]	−−	+	+	0	0	0
5.	[17]	−−	++	−−	0	0	++
6.	[12], [13]	−	+	++	0	−	−−
7.	[50]	+	+	++	0	−	+
8.	[36]	−−	++	++	0	0	++
9.	[30]	+	++	+	−−	+	+
10.	[11], [10]	++	++	++	−−	++	++
11.	[24], [23]	+	++	+	0	−−	0
12.	[22]	−−	++	0	0	−−	0
	diese Arbeit	++	+	++	++	++	++

Tabelle 8.1.: Einordnung dieser und der verwandten Arbeiten aus Kapitel 2.

Der vorgestellte Ansatz wurde im Kontext Service-orientierter Softwaresysteme entwickelt. Der Ansatz müsste sich auch auf komponenten-basierte Softwaresysteme anwenden lassen. Diese sind persistent und zustandsbehaftet und damit ändert sich nichts am Abstraktionsverfahren.

Eine weitere Fragestellung, die sich aus dieser Arbeit ergeben hat, betrifft die Assoziativität und Kommutativität des parallelen Operators „ ‖ “ der (G,G)-PRS. Das Gegenbeispiel 6.1 eines nicht einfach asynchronen Service-orientierten Systems in Abschnitt 6.4 zeigt ein System mit asynchronen Prozeduren, die sich gegenseitig rekursiv aufrufen. Durch die Assoziativität des „ ‖ “-Operators wird dieser Deadlock in der (G,G)-PRS-basierten Abstraktion und in der k-kontext-sensitiven Abstraktion nicht erkannt. Andererseits wird die Assoziativität und Transitivität bei einfach asynchron aufgerufenen Prozeduren benötigt, um diese zu synchronisieren. Eine Möglichkeit wäre die Erweiterung der (G,G)-PRS um einen zusätzlichen „ ‖ “-Operator, der nicht assoziativ ist. Dabei stellt sich die Frage, ob durch die Einführung eines nicht assoziativen Operators das Erreichbarkeitsproblem entscheidbar bleibt und welche Auswirkungen diese auf verwendbare Werkzeuge hat.

Literaturverzeichnis

[1] Aalst, Wil M. d.: Verification of workflow nets. In: *International Conference on Application and Theory of Petri Nets* Springer, 1997, S. 407–426

[2] Aalst, Wil M. d.: Workflow verification: Finding control-flow errors using petri-net-based techniques. In: *Business Process Management*. Springer, 2000, S. 161–183

[3] Aho, Alfred V. ; Lam, Monica S. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006. – ISBN 0321486811

[4] Allen, Robert ; Garlan, David: A formal basis for architectural connection. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6 (1997), Nr. 3, S. 213–249

[5] Ashton, Kevin u. a.: That 'internet of things' thing. In: *RFID journal* 22 (2009), Nr. 7, S. 97–114

[6] Balzert, Helmut: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Springer-Verlag, 2011

[7] Baraiya, V ; Singh, V: Netflix Conductor: A microservices orchestrator. In: *Netflix* 12 (2016), Nr. 59, S. 2–12

[8] Boer, Frank S. ; Bravetti, Mario ; Grabe, Immo ; Lee, Matias ; Steffen, Martin ; Zavattaro, Gianluigi: A petri net based analysis of deadlocks for active objects and futures. In: *International Workshop on Formal Aspects of Component Software* Springer, 2012, S. 110–127

[9] Böhmer, Matthias ; Hecht, Brent ; Schöning, Johannes ; Krüger, Antonio ; Bauer, Gernot: Falling Asleep with Angry Birds, Facebook and Kindle: A Large Scale Study on Mobile Application Usage. In: *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*. New York, NY, USA : ACM, 2011 (MobileHCI '11). – ISBN 978–1–4503–0541–9, 47–56

[10] Both, Andreas: *Protocol Conformance Checking of Component-based Systems and*

Service-oriented Architectures, Martin Luther University Halle-Wittenberg, Nat. Fak. III, Institute for Computer Science, Diss., 2010

[11] Both, Andreas ; Zimmermann, Wolf: Automatic Protocol Conformance Checking of Recursive and Parallel Component-Based Systems. In: *Component-Based Software Engineering, 11th International Symposium (CBSE 2008)*, 2008, S. 163–179

[12] Bouajjani, Ahmed ; Echahed, Rachid ; Habermehl, Peter: Verifying infinite state processes with sequential and parallel composition. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* ACM, 1995, S. 95–106

[13] Bouajjani, Ahmed ; Emmi, Michael: Analysis of recursively parallel programs. In: *ACM Sigplan Notices* Bd. 47 ACM, 2012, S. 203–214

[14] Breivold, Hongyu P. ; Larsson, Magnus: Component-based and service-oriented software engineering: Key concepts and principles. In: *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)* IEEE, 2007, S. 13–20

[15] Burkart, Olaf ; Steffen, Bernhard: Model checking for context-free processes. In: *CONCUR'92* Springer, 1992, S. 123–137

[16] Burkart, Olaf ; Steffen, Bernhard: Pushdown processes: Parallel composition and model checking. In: *CONCUR'94: Concurrency Theory*. Springer, 1994, S. 98–113

[17] Camilli, Matteo ; Bellettini, Carlo ; Capra, Lorenzo ; Monga, Mattia: A formal framework for specifying and verifying microservices based process flows. In: *International Conference on Software Engineering and Formal Methods* Springer, 2017, S. 187–202

[18] Cogumbreiro, Tiago ; Hu, Raymond ; Martins, Francisco ; Yoshida, Nobuko: Dynamic deadlock verification for general barrier synchronisation. In: *ACM SIGPLAN Notices* Bd. 50 ACM, 2015, S. 150–160

[19] Dahl, Ole-Johan ; Nygaard, Kristen: SIMULA: an ALGOL-based simulation language. In: *Communications of the ACM* 9 (1966), S. 671–678

[20] eMarketer: *Business of Apps. n.d. Prognose zur Anzahl der Smartphone-Nutzer weltweit von 2012 bis 2021 (in Milliarden)*. Statista, Verfügbar unter https://de.statista.com/statistik/daten/studie/309656/umfrage/prognose-zur-anzahl-der-smartphone-nutzer-weltweit/, Zugriff am 16. September 2019

[21] Falaki, Hossein ; Mahajan, Ratul ; Kandula, Srikanth ; Lymberopoulos, Dimitrios ; Govindan, Ramesh ; Estrin, Deborah: Diversity in smartphone usage. In: *Proceedings of the 8th international conference on Mobile systems, applications, and services* ACM, 2010, S. 179–194

[22] Flores-Montoya, Antonio E. ; Albert, Elvira ; Genaim, Samir: May-happen-in-parallel based deadlock analysis for concurrent objects. In: *Formal Techniques for Distributed Systems*. Springer, 2013, S. 273–288

[23] Giachino, Elena ; Kobayashi, Naoki ; Laneve, Cosimo: Deadlock analysis of unbounded process networks. In: *International Conference on Concurrency Theory* Springer, 2014, S. 63–77

[24] Giachino, Elena ; Laneve, Cosimo: A beginners guide to the deadLock Analysis Model'. In: *International Symposium on Trustworthy Global Computing* Springer, 2012, S. 49–63

[25] Gray, Jim: Why do computers stop and what can be done about it? In: *Symposium on reliability in distributed software and database systems* Los Angeles, CA, USA, 1986, S. 3–12

[26] Haddad, Serge ; Poitrenaud, Denis: Theoretical aspects of recursive Petri nets. In: *International Conference on Application and Theory of Petri Nets* Springer, 1999, S. 228–247

[27] Harrison, Nick: An Introduction to Roslyn. In: *Code Generation with Roslyn*. Springer, 2017, S. 35–53

[28] Hauck, Erwin A. ; Dent, Benjamin A.: Burroughs' B6500/B7500 stack mechanism. In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference* ACM, 1968, S. 245–251

[29] Heike, Christian ; Zimmermann, Wolf ; Both, Andreas: On expanding protocol conformance checking to exception handling. In: *Service Oriented Computing and Applications* 8 (2014), Nr. 4, S. 299–322

[30] Hicheur, Awatef ; Dhieb, Amel B. ; Barkaoui, Kamel: Modelling and analysis of flexible healthcare processes based on algebraic and recursive Petri nets. In: *Foundations of Health Information Engineering and Systems*. Springer, 2012, S. 1–18

[31] Hopcroft, John E. ; Motwani, Rajeev ; Ullman, Jeffrey D.: Introduction to Automata Theory, Languages, and Computation, 2Nd Edition. In: *SIGACT News*

32 (2001), März, Nr. 1, 60–65. http://dx.doi.org/10.1145/568438.568455. – DOI 10.1145/568438.568455. – ISSN 0163–5700

[32] Jones, Simon L. ; Ferreira, Denzil ; Hosio, Simo ; Goncalves, Jorge ; Kostakos, Vassilis: Revisitation analysis of smartphone app use. In: *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing* ACM, 2015, S. 1197–1208

[33] Karp, Richard M. ; Miller, Raymond E.: Parallel program schemata: A mathematical model for parallel computation. In: *8th Annual Symposium on Switching and Automata Theory (SWAT 1967)* IEEE, 1967, S. 55–61

[34] Klein, Gerwin ; Andronick, June ; Fernandez, Matthew ; Kuz, Ihor ; Murray, Toby ; Heiser, Gernot: Formally verified software in the real world. In: *Communications of the ACM* 61 (2018), Nr. 10, S. 68–77

[35] Langmaack, Hans: Friedrich L. Bauers und Klaus Samelsons Arbeiten in den 1950er-Jahren zur Einführung der Begriffe Kellerprinzip und Kellerautomat. In: *Keller, Stack und automatisches Gedächtnis*, 2014, S. 19–29

[36] Martens, Axel: Analyzing web service based business processes. In: *International Conference on Fundamental Approaches to Software Engineering* Springer, 2005, S. 19–33

[37] Mattern, Friedemann ; Flörkemeier, Christian: Vom Internet der Computer zum Internet der Dinge. In: *Informatik-Spektrum* 33 (2010), Nr. 2, S. 107–121

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

[39] Mayr, Richard: Process rewrite systems. In: *Information and Computation* 156 (2000), Nr. 1-2, S. 264–286

[40] Mayr, Richard: Process Rewrite Systems. In: *Information and Computation* 156 (2000), Nr. 1-2, S. 264–286

[41] Naik, Mayur ; Park, Chang-Seo ; Sen, Koushik ; Gay, David: Effective static deadlock detection. In: *Proceedings of the 31st International Conference on Software Engineering* IEEE Computer Society, 2009, S. 386–396

[42] Namiot, Dmitry ; Sneps-Sneppe, Manfred: On micro-services architecture. In: *International Journal of Open Information Technologies* 2 (2014), Nr. 9, S. 24–27

[43] Nierstrasz, Oscar: Regular types for active objects. In: *OOPSLA* Bd. 93, 1993, S. 1–15

[44] Petri, Carl A.: *Kommunikation mit automaten*, Technical University Darmstadt, Diss., 1962

[45] Petri, Carl A.: Introduction to general net theory. In: *Net theory and applications*. Springer, 1980, S. 1–19

[46] Plasil, Frantisek ; Visnovsky, Stanislav: Behavior protocols for software components. In: *IEEE transactions on Software Engineering* 28 (2002), Nr. 11, S. 1056–1076

[47] Reisig, Wolfgang: *Petrinetze: modellierungstechnik, analysemethoden, fallstudien*. Springer-Verlag, 2010

[48] Reisig, Wolfgang: *Petri nets: an introduction*. Bd. 4. Springer Science & Business Media, 2012

[49] Schwoon, Stefan: *Moped - a model-checker for pushdown systems*. 2002

[50] Seghrouchni, A El F. ; Haddad, Serge: A recursive model for distributed planning. In: *Proceedings of the 2nd International Conference on Multi-Agent Systems (ICMAS'96)*, 1996, S. 307–314

[51] Staroletov, Sergey ; Dubko, Anatoliy: A Method to Verify Parallel and Distributed Software in C# by Doing Roslyn AST Transformation to a Promela Model. In: *System Informatics, No. 15 (2019)*

[52] Tenzer, Jennifer ; Stevens, Perdita: Modelling recursive calls with UML state diagrams. In: *International Conference on Fundamental Approaches to Software Engineering* Springer, 2003, S. 135–149

[53] Verbeek, Henricus M. ; Basten, Twan ; Aalst, Wil M. d.: Diagnosing workflow processes using Woflan. In: *The computer journal* 44 (2001), Nr. 4, S. 246–279

[54] Weissbach, Mandy ; Zimmermann, Wolf: On Abstraction-Based Deadlock-Analysis in Service-Oriented Systems with Recursion. In: *European Conference on Service-Oriented and Cloud Computing* Springer, 2017, S. 168–176

[55] Wilhelm, Reinhard: Keller im Übersetzerbau. In: *Keller, Stack und automatisches Gedächtnis*, 2014, S. 43–53

[56] Zimmermann, Wolf ; Schaarschmidt, Michael: Automatic Checking of Compo-

nent Protocols in Component-Based Systems. In: LÖWE, Welf (Hrsg.) ; SÜDHOLT, Mario (Hrsg.): *Software Composition* Bd. 4089, Springer, 2006 (LNCS), S. 1–17

A. Anhang

A.1. Beispielsprache $\mathcal{XYZ}$

Im Anhang A wird die Beispielsprache $\mathcal{XYZ}$ vorgestellt. Es wird eine Syntax eingeführt und ein kleines Beispiel eines Service-orientierten Systems vorgestellt. Die Sprache steht prototypisch für andere Sprachen.

A.1.1. Beispiel - Service-orientiertes Softwaresystem

In Beispiel A.1 ist ein Service-orientiertes System nach der in diesem Kapitel beschriebenen Syntax zu sehen. Das System besteht aus fünf Diensten und vier Schnittstellen und deren Beschreibung.

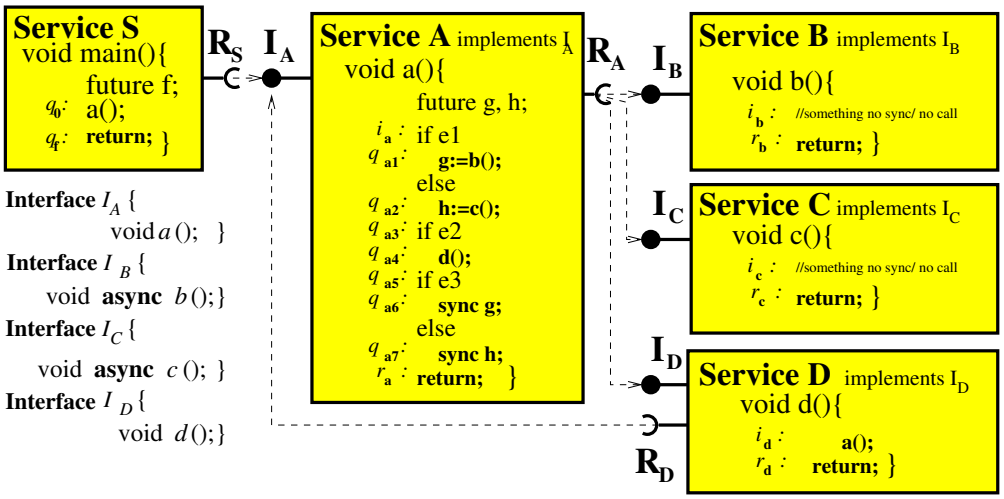


Abbildung A.1.: Ein service-orientiertes System bestehend aus den Diensten S, A, B, C und D mit den Signaturen für die Schnittstellen I_A, I_B, I_C und I_D.

A.1.2. Programmstruktur

Ein Service-orientiertes Softwaresystem besteht im Grunde aus zwei Teilen. Der erste Teil definiert die Schnittstellen. Ein Service-orientiertes System kann ein oder mehrere Schnittstellen zur Verfügung haben. Im zweiten Teil wird die Syntax der implementierten Dienste definiert.

```
1  Prog ::= Interfaces Services
2  Interfaces ::= Interfaces' Interface
3  Interfaces' ::= Interfaces' Interface
4  Interfaces' ::=
5
6  Services ::= Services' Service
7  Services' ::= Services' Service
8  Services' ::=
```

Listing A.1: Syntax zur allgemeinen Programmstruktur.

Neben der Definition wird auch festgelegt, ob eine Prozedur synchron oder asynchron aufgerufen wird. Die Syntax zur Definition der Schnittstellen beschreibt des Listing A.2 der Beispielsprache $\mathcal{XYZ}$.

```
1  Prog ::= Interfaces Services
2  Interfaces ::= Interfaces' Interface
3  Interfaces' ::= Interfaces' Interface
4  Interfaces' ::=
5  Interface ::= "Interface" Name "{" Procs "}"
6  Procs ::= Procs' Proc | Proc
7  Procs' ::= Proc'
8  Procs' ::=
9  Proc ::= Type Proctype Name "("VarType identifier ("," VarType
       identifier)*")"? ";"
10   | Type Name "("VarType identifier ("," VarType identifier)*")"?
         ";"
11 VarType ::= identifier
12 Type ::= "void" | VarType
13 Proctype ::= "async"
14 Name ::= identifier
```

Listing A.2: Syntax für die Schnittstellenbeschreibung.

Ein Service-orientiertes Softwaresystem besteht aus mehreren Services (vgl. Listing A.3).

```
1  Services ::= Services' Service
2  Services' ::= Services' Service
3  Services' ::=
4  Service ::= "Service" Name "{" MBlock "}"
5  Service ::= "Service" Name "implements" Is "{" Block "}"
```

```
Is ::= Is' I
Is' ::= Is' Is
Is ::=
I ::= Name
MBlock ::= Decls MProcedure Procedures
Decls ::= Decls' Decl
Decls' ::= Decls' Decl
Decls' ::=
MProcedure ::= "void" "main" "(" ")" "{" Block "return" ";" "}"
Procedures ::= Procedures' Procedure
Procedures' ::= Procedures' Procedure
Procedures' ::=
Procedure ::= Type Name "("VarType identifier ("," VarType
   identifier)")"? "{" PBlock "return" ";" "}"
Block ::= Decls Procedures
PBlock ::= Decls Stats
Stats ::= Stats' ";" Stat
Stats' ::= Stats' ";" Stat
Stats' ::=
```

Listing A.3: Programmstruktur.

Grundsymbole

```
keyword ::= 'while'|'if'| 'else'|'return'| 'void'|'implements'|'
   call'|'Service'|'implements'|'sync'|'async'.
special ::= '+'|'-'|'='|'=='.
digit ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'.
letter ::= 'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'
   |'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'|'a'|'b'|'c'|'d
   '|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'
   t'|'u'|'v'|'w'|'x'|'y'|'z'.
integer ::= digit (digit)*.
identifier ::= letter(letter|digit)*.
```

Listing A.4: Grundsymbole.

Vordefinierte Bezeichner

Die vordefinierten Bezeichner werden in Tabelle A.1 angegeben.

Variablendeklarationen

In dieser Programmiersprache können nur Variablen vom Typ Integer, Boolean oder Futures definiert werden (vgl. Tab. A.1).

Bezeichner	Bedeutung
integer	Integer Typ.
boolean	Logischer Typ.
false	Falschheit.
true	Wahrheit.

Tabelle A.1.: Vordefinierte Bezeichner.

```
Decl ::= VarDecl
VarDecl ::= VarType Name ";"
```

Listing A.5: Variablendeklarationen.

Anweisungen und Ausdrücke

Im Listing A.6 wird die Syntax für Anweisungen und Ausdrücke gezeigt. Es gibt Variablen vom Typ *integer*, *Bool* und *Future*.

```
Stat ::= Expr | Iteration | Assign | Sync | ProcC | Skip
Expr ::= AExpr | BExpr
AExpr ::= Name | Sum
BExpr ::= "true" | "false" | not BExpr | BExpr "<=" BExpr | BExpr
    "&&" BExpr

Iteration ::= "while" Expr "{" Stats "}"
Assign ::= Name ":=" Rightside ";"
Rightside ::= intconst | identifier | ProcC
Cond ::= "if" Expr "{" Stats "}" "else" "{" Stats "}" | "if" Expr
    "{" Stats "}"
Sync ::= "sync" Name ";"
ProcC ::= Name "(" ")" ";"| Name "(" argument ")" ";"
argument ::= Expr
Name ::= identifier
Sum ::= Sum AddOp Term
Sum ::= Term
AddOp ::= "+" | "-"
Expr ::= identifier
```

Listing A.6: Anweisungen und Ausdrücke in der Beispielsprache $\mathcal{XYZ}$.

Auswertung von Ausdrücken

Jeder Ausdruck bezeichnet ein Element einer Datenstruktur. Wenn ein Ausdruck Variablen enthält, dann hängt der Wert des Ausdrucks vom Inhalt der Variable ab. So wird die Berechnung dieses Wertes eines Ausdrucks x mit $eval(x)$ gekennzeichnet. Diese Funktion $eval : EXPR \rightarrow VAL$ wird induktiv definiert durch:

- v ist Variable, so ist $eval(v)$ der in v gespeicherte Wert
- jede Konstante *const* ist bereits ein Wert und damit ist $eval(const) = const$.
- sind $y_1, ..., y_n$ die Werte der Ausdrücke $s_1, ..., s_n$ und $f(s_1, ..., s_n)$ der zusammengesetzte Ausdruck , so gilt $eval(f(s_1, ..., s_n)) = f(eval(s_1), ..., eval(s_n))$.

Bemerkung A.1. *Die vorgestellte Sprache steht prototypisch für andere Sprachen.*

A.1.3. Semantik

Schlüsselwort	Semantik
kein Angabe	Synchroner Funktionsaufruf.
async	Funktionsaufruf erfolgt asynchron.
sync	Asynchron aufgerufene Funktion wird synchronisiert.

Futures

In der Beispielsprache soll der Aufruf einer jeden asynchronen Prozedur oder Funktion protokolliert werden. Aus diesem Grund werden Futures eingeführt.

```
1 Stat  ::= Expr | Iteration | Assign | Sync | ProcC | Skip | Future
2 ...
3 It  ::= "while" Expr "{" Stats "}"
4 Cond  ::= "if" Expr "{" Stats "}" "else" "{" Stats "}"
5     | "if" Expr "{" Stats "}"
6 ProcC   ::= Name "(" ")" ";"
7     |Name ":=" Name "(" argument ")" ";"
8 Sync  ::= "sync" Name ";"
```

Listing A.7: Deklarationen von Futures.

Schnittstellen

Ob eine benötigte oder zur Verfügung gestellte Prozedur oder Funktion asynchron ist, wird in der entsprechenden Schnittstellenbeschreibung des Services beschrieben. Die Schnittstellenbeschreibungen enthalten die Signatur einer Prozedur oder Funktion sowie das Schlüsselwort *async*. Der Aufruf oder die Benutzung dieser Prozedur oder Funktion findet damit asynchron statt (vgl. Def. 3.10, 3.9 und 3.7).

A.2. Process Rewrite System

A.2.1. PRS Hierarchie und Klassifikation

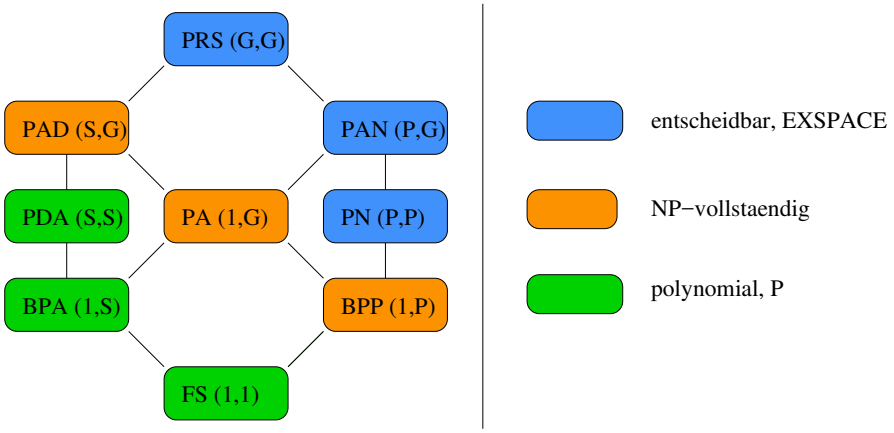


Abbildung A.2.: Mayr's PRS-Hierarchie inklusive der Komplexitätsklassen bezüglich der Erreichbarkeitsanalyse [38].

In Abbildung A.2 wird die PRS-Hierarchie nach Mayr gezeigt. Die PRS Klasse eines Prozessersetzungssystem richtet sich nach dem Aufbau der Prozessterme auf der linken und rechten Seite der Regeln.
Prozessterme werden in nach vier Klassen unterteilt [38]:

1 Terme bestehen aus einer einzelnen Variable, z.B. q_0

S Terme bestehen aus einer einzelnen Variable oder einer sequentiellen Komposition von Prozessvariablen, z.B. $q_2.q_1.q_0$

P Terme bestehen aus einer einzelnen Variable oder einer parallelen Komposition von

Prozessvariablen, z.B. $q_2 \parallel q_1 \parallel q_0$

G Allgemeine Prozessterme bestehen aus beliebigen sequentiellen und parallelen Kompositionen, z.B. $(q_2 \parallel q_1).q_0$

Sei $\alpha, \beta \in \{1, S, P, G\}$, dann ist demnach ein (α, β)-PRS eine endliche Menge an Termersetzungsregeln der Form $l \to r$, wobei der Term l aus α ist und $l \neq \epsilon$ und der Term r ist aus β.

PRS Klasse	Abkürzung	Modell
(1,1)	FS	Endliche Automaten
(1,S)	BPA	Kontext-freie Systeme
(1,P)	BPP	Einfache parallele Prozesse
(S,S)	Pushdown Processes	Kellersysteme.
(1,G)	PA-Processes	Prozessalgebren
(P,P)	Petri Nets	Petri-Netze
(S,G)	PAD	Kleinste gemeinsame Generalisierung von Kellersystemen und Prozessalgebren
(P,G)	PAN	Kleinste gemeinsame Generalisierung von **P**rozess**a**lgebren und **P**etri-**N**etze.
(G,G)	PRS	Process Rewrite Systems

Tabelle A.2.: Mayr's PRS-Hierarchie ([39], S.267).

B. Anhang

B.1. Carnegiea gigantea

Abbildung B.1.: Wild wachsende *Saguaros* an der I-17 in Phoenix, AZ.

B.2. Notation

Notation	Beschreibung
σ	Speicher, Def. 4.1, partielle Funktion $VAR \rightarrow VALUE$
σ_0	leere Speicher, nach Def. 4.1 ($= \emptyset$)
$\sigma\|_x^{val}$	Änderung der Funktion σ an der Stelle x zu val
Φ	globaler Zähler für die eindeutige Zuordnung der future
$\|\|$	Parallele Operator
.	Sequentielle Operator
Π_S	Implementierung eines Service S
$\mathcal{D}$	Platzhalterfunktion
$\rightarrow_S$	Regelmenge eines Zustandübergangssystems S
$[\![]\!]$	Semantik
α	Abstraktionsfunktion
κ	Aufrufketten
$\mathcal{I}_S$	Menge der Angebotsschnittstellen eines Services S
$\mathcal{R}$	Menge der Nutzungsschnittstellen eines Services S
sig	Signatur
$\mathcal{B}_{SoS}$	Menge der Bindungen eines Service-orientierten Systems SoS
μ	Markierungsvektor
μ_0	Heimatmarkierung

Tabelle B.1.: Überblick über die in dieser Arbeit verwendeten Notationen.

Eidesstattliche Erklärung/ Declaration under Oath

Ich erkläre an Eides statt, dass ich die Arbeit selbstständig und ohne fremde Hilfe verfasst, keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommen Stellen als solche kenntlich gemacht habe.

I declare under penalty of perjury that this thesis is my own work entirely and has been written without any help from other people. I used only the sources mentioned and included all the citations correctly both in word or content.

.....	.....
Datum/	Unterschrift des Antragstellers/
Date	Signature of the applicant

Lebenslauf

Persönliche Daten

Name, Vorname:	Weißbach, Mandy
Geburtsname:	Günther
Email-Adresse:	mandy.weissbach@informatik.uni-halle.de
Geburtsdatum:	13. März 1983
Geburtsort:	Querfurt
Staatsangehörigkeit:	deutsch
Familienstand:	verheiratet, zwei Kinder

Schul- und Hochschulausbildung

10/2003 - 09/2009	Studentin der Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik Studiengang: Diplom Bioinformatik Schwerpunkte: Datenbanken und Informationssysteme, Biochemie Abschluss: Diplom-Bioinformatikerin
06/2002 - 07/2003	AuPair in Phoenix, AZ USA, Studentin am Mesa Community College, Kurs: English as a Second Language, Level 4
1993-2002	Gymnasium in Querfurt Abschluss: Abitur

Berufliche Stationen

seit 12/2009	Mitarbeiterin der Martin-Luther-Universität Halle-Wittenberg Lehrstuhl für Software-Engineering und Programmiersprachen, Prof. Dr. Wolf Zimmermann, Tätigkeit in Forschung und Lehre **Unterbrechungen:** • 11/2013 - 07/2015 Mutterschutz und Elternzeit • 04/2011 - 03/2012 Mutterschutz und Elternzeit
08/2009 - 11/2009	Mitarbeiterin am IPK Gatersleben, AG Bioinformatik und Informationstechnologie, Integration einer funktionalen Textindizierungsinfrastruktur in die bestehende IPK Life Science Suchmaschine LAILAPS
01/2009 - 07/2009	Studentische Hilfskraft am IPK Gatersleben, AG Bioinformatik und Informationstechnologie, Konzeption und Implementierung einer Textindizierungsinfra- struktur zur Performanceoptimierung von Suchmaschinen in den Lebenswissenschaften
10/2007 - 07/2008	Praktikum bei der Firma Kapelan Bio-Imaging, Konzeption und Entwicklung von LabImage CC Colony Counting