TU Clausthal

Benjamin Fischer

Verfahren zur dezentralen anwendungsspezifischen Verschaltung

von Komponenten zu einer optimalen Auswahl von Applikationen

SSE-Dissertation 13

Software Systems Engineering

Institut für Informatik
Lehrstuhl von Prof. Dr. Andreas Rausch

Verfahren zur dezentralen anwendungsspezifischen Verschaltung

von Komponenten zu einer optimalen Auswahl von Applikationen

D i s s e r t a t i o n

zur Erlangung des Grades
Doctor rerum naturalium
(Dr. rer. nat.)

vorgelegt von

Benjamin Fischer
aus Salzgitter

genehmigt vom Institut für Informatik
der Technischen Universität Clausthal

2016

Dissertation Clausthal, SSE-Dissertation 13, 2016

Vorsitzende(r) der Promotionskommission
Prof. Dr. Michael Kolonko

Hauptberichterstatter/in
Prof. Dr. Andreas Rausch

2. Berichterstatter/in
Prof. Dr. Jörg P. Müller

Tag der mündlichen Prüfung: September 19, 2016

Kurzbeschreibung

Software ist mittlerweile ein fester Bestandteil unseres täglichen Lebens geworden. Ob wir auf dem Handy kurz die aktuelle Wetterprognose prüfen oder unser Auto eine Route plant, ständig nutzen wir dazu Softwareanwendungen. Daher besteht auch ein großer Anspruch, dass uns diese Software ständig zur Verfügung steht und zuverlässig arbeitet. Für ein zuverlässiges Arbeiten der Software werden meist viele Informationen benötigt, welche sich die Software beschaffen muss, wofür sie sich mit anderen Softwaresystemen vernetzen muss.

Bei der Vielfalt der aktuellen Softwareanwendungen ist es nicht mehr möglich, das gesamte Softwaresystem, auf dem die Anwendungen laufen, als Ganzes zu entwickeln. Stattdessen werden offene verteilte Softwaresysteme erforscht, in denen die einzelnen Anwendungen zur Designzeit nicht bekannt sind. Außerdem entstehen immer mehr autonome Anwendungen, welche sich die von ihnen benötigten Informationen selbst beschaffen. Die Erforschung verteilter Systeme mit autonomen Anwendungen bringt viele Herausforderungen mit sich. Softwaresysteme sind heutzutage häufig direkt an Hardware gekoppelt, wie z. B. in Fernsehern, Kühlschränken oder Autos. Daher können diese Softwaresysteme ihre Dienste nur so oft anbieten, wie die entsprechende Hardware diese auch bearbeiten kann. Eine Herausforderung in verteilten Systemen besteht in einer, wenn möglich, optimalen Verteilung solch beschränkter Ressourcen.

In dieser Arbeit wird ein Verfahren präsentiert, welches eine optimale Menge an Anwendungen aus Softwareteilen, sogenannte Komponenten, erstellt. Aufgrund der Existenz autonomer Anwendungen oder Komponenten wird die Verteilung dezentral erzeugt.

Eingebaut in eine entsprechende Softwareinfrastruktur gewährleistet dieses Verfahren die optimale Verschaltung von Komponenten zu einer Menge von Anwendungen. Das Verfahren arbeitet dezentral zur Laufzeit, enthält aber Stellgrößen, welche zentral vorgegeben werden, um das Verfahren auf die entsprechende Umgebung anzupassen.

Danksagung

Mein herzlicher Dank gilt allen, die mir direkt oder indirekt bei der Erstellung dieser Arbeit geholfen haben.

An erster Stelle möchte ich mich bei meinem Betreuer Prof. Dr. Andreas Rausch für seine intensive Betreuung bedanken. Über interessante Projekte an seinem Lehrstuhl habe ich mir die Problemstellung erarbeiten können. Anschließend hat er mir mit Anregungen, kritischen Kommentaren und vielen Diskussionen bei der Erstellung dieser Dissertation sehr geholfen.

Auch geht mein Dank an Prof. Dr. Jörg P. Müller für seine Bereitschaft als Zweitgutachter zu fungieren und für seine hilfreichen Kommentare.

Auch bei meiner Familie und Freunden möchte ich mich bedanken. Sie standen mir die gesamte Zeit besonders in den stressigen Zeiträumen liebevoll und unterstützend zur Seite.

Des Weiteren möchte ich mich bei meinen Kollegen Falk Howar, Marcel Ibe, Christoph Knieke und Mirco Schindler für die teils sehr ausufernden und hilfreichen Diskussionen bedanken.

Abschließend möchte ich mich noch bei allen Kollegen und Angestellten am Lehrstuhl von Prof. Dr. Andreas Rausch für die vielen anregenden und konstruktiven Gespräche bedanken.

Inhaltsverzeichnis

1 Einleitung

Softwaresysteme sind heutzutage ein fester Bestandteil unseres Lebens geworden – ob wir mit unserem Smartphone telefonieren oder eine SMS schreiben, ob wir mit dem Auto zum Einkaufen fahren oder unsere Waschmaschine einstellen. In fast jedem Gerät ist mittlerweile Software integriert, welche wir durch die Nutzung des Gerätes verwenden.

Aufgrund dieser Tatsache, dass wir Softwaresysteme in unserem Alltag immer stärker nutzen, steigt auch unser Anspruch an mehr Funktionalität dieser Softwaresysteme. So können wir mit dem Smartphone u. a. beim Telefonieren den Gesprächspartner sehen, E-Mails lesen und verfassen, sich über aktuelle Nachrichten aus der ganzen Welt informieren und vieles mehr. Um die ansteigende Anzahl an Funktionalitäten anbieten zu können, werden allerdings immer komplexere Softwaresysteme benötigt.

Gleichzeitig steigt der Grad der Vernetzung der Softwaresysteme untereinander. Damit die einzelnen Softwaresysteme ihre Funktionalitäten auch anbieten können, werden meist Informationen aus anderen Softwaresystemen benötigt. Z. B. benötigt das Smartphone zum Anzeigen der aktuellen Nachrichten eine Verbindung zu den Nachrichtenzentralen. Diese Vernetzung ist in vielen Fällen sehr dynamisch, ohne dass der Benutzer im besten Fall davon eine Notiz nehmen soll. Beim Telefonieren auf der Autobahn z. B. verbindet sich das Smartphone rechtzeitig und automatisch mit neuen Sendemasten, um dem Benutzer ein durchgehendes Telefonat zu ermöglichen. Daher werden nicht nur komplexere Softwaresysteme benötigt, sondern auch Softwaresysteme, welche sich untereinander dynamisch vernetzen können.

Die Entwicklung solcher komplexer Softwaresysteme mit dynamischer Vernetzung stellen die Softwareentwickler immer noch vor große Herausforderungen. In den folgenden Abschnitten dieses Kapitels werden die wesentlichen Herausforderungen der betrachteten Softwaresysteme zu aktuellen Trends genannt. Anschließend wird die zentrale Problemstellung dieser Arbeit formuliert. Der darauffolgende Abschnitt benennt die konkreten Ziele dieser Arbeit. Abschließend wird eine kurze Übersicht über die einzelnen Kapitel dieser Arbeit gegeben.

1.1 Ausgangssituation

Wie die folgenden Trends zeigen, wird an komplexen und verteilten Softwaresystemen aktuell intensiv geforscht. Ebenfalls werden Softwaresysteme mit begrenzten Ressourcen oder einem hohen Grad an autonomen Einheiten betrachtet. Auf Basis dessen wird anschließend der Bereich beschrieben, aus dem die Problemstellung dieser Arbeit herausgearbeitet wird.

1.1.1 Entwicklung komplexer Softwaresysteme – Trend zur Verwendung einer komponentenbasierten Middleware

Komplexe Softwaresysteme stellen Entwickler immer wieder vor eine große Herausforderung. Wie der jährliche CHAOS-Report der Standish Group (Standish Group International, Inc. 1999) jedes Jahr erneut aufzeigt, schlagen viele Softwareprojekte fehl oder überschreiten die gesetzten Grenzen an Zeit oder Kosten für das jeweilige Softwareprojekt.

Aktuell ist es immer noch schwer, Software in einer bestimmten Zeit mit vorgegebenem Aufwand und Funktionalitäten in einer guten Qualität zu entwickeln. Diese vier Aspekte werden auch als Eckpunkte des „magischen Vierecks“ bezeichnet, siehe Abbildung 1-1.

Für eine erfolgreiche Softwareentwicklung sind Requirements Engineering und Architekturentwurf zwei zentrale Faktoren (Nuseibeh 2001). Fehlentscheidungen aufgrund unvollständiger Anforderungen oder eine ungeeignete Architektur zu Anfang eines Projekts haben für das zu entwickelnde Softwaresystem weitreichende und gravierende Auswirkungen. Daher ist eine geeignete Softwarearchitektur ein entscheidender Faktor für eine erfolgreiche Softwareentwicklung (Gharbi et al. 2015).

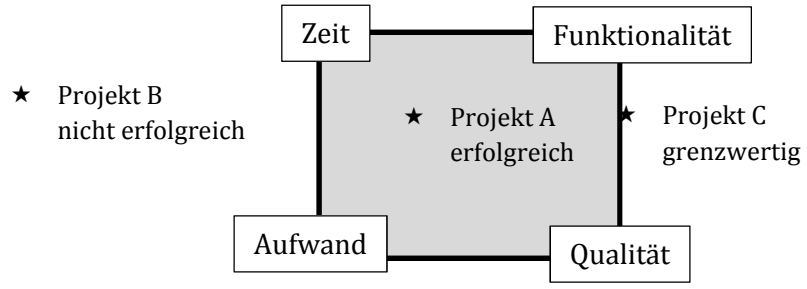

Abbildung 1-1: Das magische Viereck

Quelle: (Gharbi et al. 2015), Seite 11

Komplexe Softwaresysteme benötigen neben einer guten Architektur auch eine Sammlung von standardisierter Unterstützungssoftware, wie z. B. Middleware. Komplexe Softwaresysteme bestehen aus vielen Bauteilen (Komponenten), welche meist auf mehreren Rechnern verteilt sind. Daher wird zum einen eine Softwarearchitektur benötigt, um diese Komponenten beschreiben zu können, z. B. eine komponentenbasierte Architektur. Zum anderen existiert eine Reihe von Standardsoftware, welche z. B. Dienste für die Kommunikation von Softwarekomponenten über Rechnergrenzen bereitstellt. Java RMI (Downing 1998) und CORBA (Siegel 2000) sind zwei Beispiele dafür. Beide ermöglichen die direkte Kommunikation von Komponenten, welche sich auf verschiedenen Rechnern befinden. Solch eine Standardsoftware befindet sich in der Architektur der Software zwischen dem Betriebssystem und der zu entwickelnden Software und wird daher Middleware genannt (Bernstein 1996). Eine Middleware kapselt demnach Funktionalitäten, welche die Softwareentwickler dann nicht mehr selbst entwickeln müssen. Dies spart Aufwand und Zeit bei der Entwicklung. Zusätzlich wurde diese Middleware meist schon ausgiebig geprüft, weshalb bei dieser von einer guten Qualität ausgegangen werden kann. Daher unterstützt die Verwendung einer passenden Middleware das erfolgreiche Entwickeln einer Software anhand des „magischen Vierecks“.

Neben Diensten bieten manche Middleware-Lösungen auch ein Komponentenmodell an (Rütschlin et al. 2001). Ein Komponentenmodell beschreibt den Aufbau einer Komponente und die Interaktion zwischen Komponenten. Mittels eines gemeinsamen Komponentenmodells können unterschiedliche Anwendungen direkt miteinander interagieren. Eine weitere Funktionalität ist die dynamische Adaptivität eines Softwaresystems bezogen auf ihre Komponenten. Durch die Standardisierung der Komponentenstruktur kann die Verschaltung der Komponenten von der Middleware übernommen werden. Je nach Middleware kann sich diese Verschaltung auch dynamisch anpassen. Wenn eine Komponente ausfällt, kann automatisch nach Ersatz gesucht werden. Bei Verwendung solch einer Middleware müssen Softwareentwickler nur noch die einzelnen Komponenten entwickeln und eine Struktur für die Verschaltung konstruieren.

1.1.2 Internet der Dinge – Trend zur hardwarebezogenen Entwicklung von Software

Das Internet der Dinge (Internet of Things) gewinnt heutzutage immer mehr an Bedeutung (Keller et al. 2012). Unter diesem Begriff wird die selbstständige Kommunikation von Systemen untereinander über das Internet verstanden. Bisher wurde das Internet überwiegend von Menschen genutzt, um z. B. Informationen zu suchen oder Dienste, wie z. B. das Onlinebanking, zu nutzen. Allerdings wird schon länger an Systemen geforscht, um unterschiedliche Systeme miteinander zu verbinden.

Schon jetzt existieren viele Anwendungsgebiete für solche Systeme. Beim „intelligenten Wohnen“ können Geräte wie Licht, Rollläden und Heizung mit unterschiedlichen Sensoren verbunden werden. Das intelligente Haus kann dann z. B. Strom einsparen, indem es das Licht in Räumen, wo sich keiner aufhält, dimmt oder ganz ausschaltet. Des Weiteren kann es die Heizung regeln, um auf evtl. Sonneneinstrahlung oder geöffnete Fenster automatisch reagieren zu können.

Auch das Auto wird durch die Vernetzung mit dem Internet zunehmend intelligenter (Keller et al. 2012). Durch die Kommunikation mit anderen Systemen wird sogar das autonome Fahren erst möglich gemacht. Schon jetzt ermöglichen Sensoren das selbstständige Einparken in einer Parklücke oder das Verringern von Auffahrunfällen durch selbstständiges Abbremsen. Beim autonomen Fahren werden vom Auto über das Internet z. B. Informationen über die aktuelle Straßenlage abgefragt oder, wie es schon in San Francisco möglich ist, ein Parkplatz wird selbstständig reserviert (Millard-Ball et al. 2014).

Wie an dem Parkplatzbeispiel deutlich nachzuvollziehen, können allerdings nur so viele Autos dort parken oder einen Parkplatz reservieren wie freie Parkplätze zur Verfügung stehen. Der Dienst eines Parkplatzes, z. B. für das Bezahlen des Parkplatzes, kann daher zur gleichen Zeit nur begrenzt oft angeboten werden. Unter dem Aspekt „Internet der Dinge“ existiert eine Vielzahl von Geräten, welche untereinander kommunizieren und einen Dienst nur begrenzt oft anbieten können. Die optimale Verteilung solcher begrenzter Ressourcen ist dabei eine große Herausforderung.

1.1.3 Autonome Systeme – Trend zum dezentralen Kollaborieren

Aufgrund der steigenden Komplexität von Software werden autonome und intelligente Systeme immer häufiger verwendet (Huebscher und McCann 2008; Nami und Sharifi 2007). Diese Systeme können wegen ihrer Komplexität nicht mehr manuell gesteuert werden. Daher interagieren viele Systeme selbstständig untereinander und mit der Umgebung. Ein Beispiel dafür sind autonome Fahrzeuge, welche aktuell intensiv entwickelt werden (Mogelmose et al. 2012; Skog und Handel 2009; Goerzen et al. 2009). Diese Fahrzeuge sollen in naher Zukunft selbstständig auf den Straßen fahren können. Dazu müssen sich diese Fahrzeuge mit anderen Systemen vernetzen, um z. B. aktuelle Informationen zur Verkehrssituation erhalten oder einen Parkplatz reservieren zu können. Aber die Fahrzeuge müssen auch untereinander kommunizieren können, um z. B. Unfälle zu vermeiden.

Autonome Systeme verwalten sich selbst auf Basis ihres lokalen Wissens, können daher nicht von einer zentralen Einheit verwaltet werden. Sie entscheiden selbst, welche Informationen oder Dienste sie benötigen und mit welchem System sie sich dabei vernetzen. Um das Verhalten dieser autonomen Systeme in einem größeren System zentral beeinflussen zu können, existieren Organisationsstrukturen, welche analog zu Organisationsstrukturen in der Gesellschaft Regeln besitzen. An diese Regeln muss sich jeder Teilnehmer halten (Jensen et al. 2016). Das Erstellen

solcher komplexer dezentraler Systeme mit autonomen Einheiten stellt Entwickler auch heute noch vor große Herausforderungen.

1.1.4 Kombination von Trends – dezentrale Verschaltung von Komponenten als begrenzte Ressource

Jeder beschriebene Trend enthält eine Herausforderung, die Potenzial zur weiteren Forschung besitzt. Ich untersuche in meiner Arbeit ein Problem aus einem Bereich, in dem alle beschriebenen Herausforderungen zusammentreffen.

Dieser Problembereich besteht aus einer Menge von autonomen Komponenten, welche zu laufenden Anwendungen verschaltet werden sollen. Die Dienste, welche die Komponenten dabei anbieten, können nur begrenzt oft gleichzeitig genutzt werden. Daher entsprechen die Komponenten in dieser Hinsicht einer begrenzten Ressource. Aufgrund der Autonomie müssen die Komponenten sich selbst zu einer oder mehreren Anwendungen verschalten. Zusätzlich besitzt jede Komponente nur das Wissen, das sie für die Erfüllung ihrer Aufgaben benötigt. Keine Komponente besitzt demnach das Wissen des gesamten Systems. Allerdings sollen auch hier im System globale Richtlinien existieren, um eine optimale Zuordnung gewährleisten zu können.

1.1.5 Anwendungsbeispiel – IT-Ökosystem

Das IT-Ökosystem ist ein Anwendungsbeispiel des beschriebenen Bereichs, welches die genannten Herausforderungen enthält. Analog zu existierenden Ökosystemen besteht ein IT-Ökosystem aus autonomen Einheiten (Menschen, Systemen und der Umgebung). Zur Erreichung ihrer persönlichen Ziele müssen diese Einheiten entweder um Ressourcen konkurrieren oder mit anderen Einheiten kollaborieren. Stabilität erhält das IT-Ökosystem durch globale Richtlinien, an die sich jede Einheit halten muss (Herold et al. 2008).

Die autonomen Einheiten aus dem IT-Ökosystem entsprechen den autonomen Komponenten aus dem beschriebenen Problembereich. In beiden Bereichen existiert Kollaboration zum Erreichen der persönlichen Ziele. Des Weiteren wird bei beiden Bereichen durch die Autonomie dezentrales Kollaborieren motiviert.

Entsprechend dem Problembereich besitzt auch das IT-Ökosystem begrenzte Ressourcen in Gestalt der autonomen Einheiten. Menschen, Systeme und die Umgebung existieren nur begrenzt in einem IT-Ökosystem. Auch können die einzelnen Einheiten nicht unbegrenzt viele Aufgaben gleichzeitig erledigen. Daher besitzt das IT-Ökosystem ebenfalls die Herausforderung der optimalen Verteilung von Ressourcen. Somit enthält dieses Anwendungsbeispiel alle Herausforderungen aus dem Problembereich.

1.1.6 Stand der dezentralen Verschaltung von Komponenten als begrenzte Ressource zu einer Menge von Anwendungen

Für die drei genannten Herausforderungen existiert, einzeln betrachtet, eine Reihe von Forschungsergebnissen, von denen eine Auswahl im Folgenden genannt wird. Auch für jeweils zwei der Herausforderungen zusammen existieren Arbeiten.

Dezentrales Arbeiten wird unter anderem im Bereich der Multiagentensysteme intensiv erforscht (Wooldridge 2009). Auch werden darin die Kollaboration mit anderen Agenten und die dabei auftretenden Herausforderungen untersucht. Des Weiteren existieren Ergebnisse im Bereich der dezentralen Verteilung von Ressourcen (Netzer et al. 2015; Sykes et al. 2011). Allerdings betrachten diese Ergebnisse nicht die dezentrale Zuordnung von Komponenten als begrenzte Ressource zu einer Menge von Anwendungen. Komponenten können mehreren Anwen-

dungen gleichzeitig zugeordnet werden und Anwendungen benötigen meist mehr als eine Komponente zum Laufen. Außerdem können sich Bedingungen von Anwendungen, welche dieselbe Komponente verwenden wollen, widersprechen. Solche komplexen Zusammenhänge wurden bei Multiagentensystemen für eine optimale Ressourcenverteilung bisher nicht betrachtet.

Auch im Bereich komponentenbasierter Verschaltung wurden mehrere Verfahren erforscht. Sowohl dezentrale als auch zentrale Verfahren wurden entwickelt (Dowling 2004). Ebenfalls wurden zentrale Verfahren zur optimalen Verschaltung von Komponenten zu Anwendungen sowie die dezentrale Verschaltung von begrenzten Komponenten zu einer Anwendung erforscht (Klus 2013). Allerdings wurde eine dezentrale Verschaltung einer Menge von begrenzten Komponenten zu einer optimalen Menge von Anwendungen bisher noch nicht betrachtet.

Aufgrund dieser Tatsache existiert im Bereich der dezentralen Verschaltung von Komponenten zu einer optimalen Menge von Anwendungen noch Forschungsbedarf.

1.2 Problemstellung

Im letzten Abschnitt wurde der Bereich beschrieben, in dem das Problem liegt, welches in dieser Arbeit bearbeitet wird. Dieser Abschnitt beschreibt das Problem mit den einzelnen Herausforderungen, für die jeweils eine Lösung gesucht wurde.

Entsprechend dem letzten Abschnitt wird das Problem in folgender Aussage zusammengefasst:

„Es existiert kein dezentrales Verfahren, welches eine Menge von Komponenten als begrenzte Ressource zu einer optimalen Menge von Anwendungen verschalten kann."

In dieser Arbeit wird daher nach einem dezentralen Verfahren geforscht, um eine Menge von Komponenten zu einer Menge von Anwendungen zu verschalten. Dabei werden die Komponenten als begrenzte Ressource angesehen, welche optimal auf die Anwendungen verteilt werden sollen.

Allerdings wirft das Problem eine Reihe von Fragen auf, welche beim Bearbeiten des Problems betrachtet werden müssen. Die erste Frage betrifft die Definition des Optimums. Es existieren mehrere Interpretationen dazu, was einer optimalen Verschaltung entspricht. Zum einen soll die maximale Anzahl an Anwendungen laufen können. Zum anderen soll eine spezielle Auswahl von Anwendungen bevorzugt behandelt werden - oder die Auslastung der einzelnen Komponenten soll so gering wie möglich sein. Bevor überhaupt an einem Verfahren gearbeitet werden kann, muss diese Frage beantwortet sein.

Anschließend muss die Frage geklärt werden, wie dezentral mit autonomen Komponenten eine optimale Verschaltung erreicht werden kann. Dafür existieren u. a. im Bereich der Spieltheorie mehrere Verfahren, welche sich mit der dezentralen Verteilung von Ressourcen beschäftigen (Wooldridge 2009). Diese werden in dieser Arbeit auf Verwendung im zu entwickelnden dezentralen Verfahren geprüft. Zusätzlich soll das Verfahren noch zwei nachvollziehbare Anforderungen erfüllen. Zum einen soll das Verfahren bei einer gegebenen Menge an Komponenten und Anwendungen terminieren. Zum anderen soll es immer genau eine optimale Verschaltung liefern. Wenn das Verfahren bei einer gegebenen Menge nicht terminiert, können die Komponenten nicht zu Anwendungen verschaltet werden. Damit würden diese Anwendungen nicht laufen können, obwohl dies evtl. möglich wäre. Der zweite Punkt betrifft die Existenz mehrerer optimaler Verschaltungen. Wenn sich das Verfahren hier nicht für eine Lösung entscheiden kann, würde es entweder abbrechen oder zwischen den Lösungen hin- und herwechseln. Bei beiden Va-

rianten erreicht das System keinen stabilen Zustand und die beteiligten Anwendungen werden dann ebenfalls nicht laufen können. Das Verfahren muss demnach auch diese Punkte berücksichtigen.

Abschließend besteht noch die Frage nach der Korrektheit des entwickelten Verfahrens bzgl. der Anforderungen. Das Verfahren muss alle gestellten Anforderungen erfüllen und muss daher anschließend auf Korrektheit evaluiert werden.

1.3 Ziele der Arbeit

Für die Bearbeitung des Problems wurden aus dem vorherigen Abschnitt vier Ziele definiert.

1. Definition einer optimalen Verschaltung.
2. Entwicklung eines dezentralen Verfahrens zum Erreichen der optimalen Verschaltung.
3. Stabilität des Algorithmus und des erreichten Optimums sicherstellen.
4. Verfahren evaluieren.

Für eine Definition einer optimalen Verschaltung von Komponenten ist es notwendig, die einzelnen Verschaltungen bewerten zu können. Dafür wird eine komponentenbasierte Architektur benötigt, aus der eine gegebene Verschaltung anschließend mit einer zu definierenden Bewertungsfunktion bewertet werden kann.

Nach Beschreibung einer optimalen Verschaltung wird ein Verfahren benötigt, um dieses Optimum dezentral erreichen zu können. Dafür ist eine gemeinsame Kommunikationsbasis für die Komponenten und Anwendungen notwendig. Die beschriebenen Trends arbeiten in einem System, in dem zur Designzeit des Systems nicht alle Komponenten und Anwendungen bekannt sind. Zum Beispiel im Bereich Internet der Dinge muss das System mit neuen Anwendungen und Komponenten umgehen können, z. B. wenn ein neuer Kühlschrank geliefert wird. Dieser Kühlschrank soll sofort mit dem System kommunizieren und sich in das System integrieren können. Zur Sicherstellung einer gemeinsamen Kommunikation ist eine komponentenbasierte Middleware notwendig, für die jede Komponente und Anwendung entwickelt wird.

Eine weitere Herausforderung besteht in der Dynamik der Infrastruktur. Speziell in dem Bereich Internet der Dinge und dem Bereich autonome Agenten wird sich die Anzahl der Teilnehmer mit der Zeit ändern. Daher muss die Middleware die Verschaltung zur Laufzeit dynamisch adaptiv erstellen und anpassen können.

Folglich gilt es nun ein geeignetes dezentrales Verfahren zu entwickeln, um das Optimum unter diesen Bedingungen zu erzeugen. Dabei hat das Verfahren die folgenden Anforderungen zu erfüllen. Bei einer gegebenen Menge von Komponenten und Anwendungen muss das Verfahren mit der Ausgabe einer optimalen Verschaltung immer terminieren. Auch muss sich das Verfahren bei der Existenz mehrerer optimaler Verschaltungen für eine optimale Verschaltung entscheiden. Es darf nicht zwischen mehreren optimalen Verschaltungen wechseln, da das Verfahren in dem Fall nicht terminiert.

Abschließend muss das entwickelte Verfahren auf das Einhalten aller Anforderungen geprüft werden. Damit wird nachgewiesen, dass das Verfahren eine Lösung des Problems darstellt.

1.4 Beitrag dieser Arbeit

Zur Erfüllung des ersten Ziels, das Ermitteln einer passenden Architektur zur Beschreibung von Komponenten und Anwendungen, wurde das Modell von Klus (Klus 2013) verwendet. Mit die-

sem Modell können Komponenten, Anwendungen und die Zuordnung der Komponenten zu den Anwendungen dargestellt werden. Da das Modell allerdings keine Informationen darüber enthält, wie Anwendungen gegeneinander bewertet werden können, wurde es entsprechend erweitert.

Für die Definition einer optimalen Verschaltung wurde ein Anwendungsbeispiel erstellt, welches aus fünf Szenarien besteht. Jedes Szenario enthält als Ausgangskonfiguration eine Menge von gestarteten Komponenten und Anwendungen und die erwartete Zielkonfiguration, also die erwartete Zuordnung der Komponenten zu den Anwendungen. Anhand dieser Informationen wurde das Modell aus (Klus 2013) erweitert und eine Bewertungsfunktion erstellt, welche für die jeweilige Zielkonfiguration die optimale Bewertung liefert. Daraus wurde anschließend die optimale Verschaltung definiert.

Für die Entwicklung eines dezentralen Verfahrens, als zweites Ziel dieser Arbeit, wurden bestehende Verfahren betrachtet, welche eine dezentrale optimale Zuordnung von begrenzten Ressourcen gewährleisten. Dafür wurden Verfahren aus dem Bereich der Multiagentensysteme betrachtet, weil dieser Bereich wie der Problembereich aus autonomen Einheiten besteht. Es wurden zwei Verfahren (Auktion und Feilschen) (Wooldridge 2009) aus diesem Bereich ermittelt und in das Verfahren dieser Arbeit integriert. Bei der Entwicklung des Verfahrens fand ebenfalls das dritte Ziel, die Einhaltung der Stabilität, Berücksichtigung.

Abschließend wurde das Verfahren implementiert und anhand der Szenarien des Anwendungsbeispiels und zweier weiterer Szenarien auf Korrektheit evaluiert. Damit wurde auch das vierte Ziel bearbeitet.

1.5 Kapitelübersicht

In Kapitel 2 dieser Arbeit werden Grundlagen zu komponentenbasierter Softwareentwicklung, selbstorganisierenden Softwaresystemen und Multiagentensystemen beschrieben. Damit wird ein grundlegendes Verständnis für das Problem und die erarbeitete Lösung geschaffen.

Im dritten Kapitel wird ein Anwendungsbeispiel beschrieben, woraus anschließend die Problemstellung erarbeitet wird. In dem Anwendungsbeispiel werden außerdem die Anforderungen an die zu erstellende Lösung dargestellt. Zusätzlich können aus den Anforderungen auch die Forschungsfragen abgeleitet werden. Abschließend wird die erarbeitete Problemstellung mit verwandten Arbeiten verglichen, um zu zeigen, dass dieses Problem noch nicht ausreichend bearbeitet wurde.

Im vierten Kapitel werden die einzelnen Forschungsfragen aus dem dritten Kapitel untersucht und zu jeder Forschungsfrage wird eine Lösung erarbeitet. Die Ergebnisse werden anschließend an dem Anwendungsbeispiel näher erläutert. Das erarbeitete Lösungsverfahren wird in Kapitel 5 ausführlich beschrieben und außerdem an dem Anwendungsbeispiel exemplarisch durchgeführt. Im sechsten Kapitel wird eine Implementierung des Lösungsverfahrens beschrieben. Anschließend folgt im darauffolgenden Kapitel eine Erläuterung zur Verwendung der Implementierung.

Im siebten Kapitel werden anschließend aus dem Anwendungsbeispiel Testfälle generiert und an der Implementierung des Lösungsverfahrens angewendet, womit die Korrektheit des Lösungsverfahrens in Bezug auf die Anforderungen des Anwendungsbeispiels gezeigt wird. Anschließend wird mit zwei weiteren Testfällen nachgewiesen, dass das Lösungsverfahren nicht nur auf das Anwendungsbeispiel beschränkt ist.

Im letzten Kapitel werden alle Kapitel noch einmal zusammengefasst beschrieben und die Lösung in Bezug zu den Forschungsfragen wird diskutiert. Außerdem werden vorhandene Schwächen des Lösungsverfahrens für weitere Forschungsmöglichkeiten beschrieben.

2 Grundlagen

In diesem Kapitel werden die Grundlagen zum besseren Verständnis der Problemstellung und der erarbeiteten Lösung beschrieben. Der Problembereich besteht, wie im vorherigen Kapitel beschrieben, aus Komponenten und Anwendungen, wobei eine Anwendung aus einer Beschreibung der für die Anwendung benötigten Komponenten besteht. Daher wird eine komponentenbasierte Architektur für die Lösung benötigt. Aus diesem Grund werden in diesem Kapitel Grundlagen zur komponentenbasierten Softwareentwicklung vermittelt.

Außerdem arbeiten die Komponenten und Anwendungen laut der Problembeschreibung autonom. Autonome Einheiten treffen eigene Entscheidungen, um ihr Ziel zu erreichen, also auch darüber, mit wem sie zur Laufzeit kollaborieren. Diese Einheiten organisieren sich zur Laufzeit selbst. Um miteinander kommunizieren zu können, wird eine entsprechende Infrastruktur benötigt. Um die Selbstorganisation der autonomen Einheiten zu ermöglichen, muss die Infrastruktur ebenfalls selbstorganisierend sein. Daher wird in diesem Kapitel auch ein Überblick über den Aufbau eines selbstorganisierenden Softwaresystems und über einige Beispiele solcher Systeme gegeben.

Da das zu suchende Verfahren eine optimale Ressourcenzuweisung von autonomen Einheiten ermöglichen soll, wurden Verfahren aus dem Bereich der Spieltheorie betrachtet. Dieser Bereich beschäftigt sich u. a. mit genau solchen Verfahren. Dabei wird eine Auswahl der existierenden Verfahren vorgestellt, welche anschließend für das Lösungskonzept dieser Arbeit verwendet werden.

2.1 Komponentenbasierte Softwareentwicklung

Bei einer komponentenbasierten Softwareentwicklung wird das System nicht als Ganzes entwickelt. Stattdessen wird das System in miteinander kommunizierende Softwarekomponenten zerlegt. Dies ermöglicht u. a. eine übersichtliche Strukturierung des Systems (Heineman und Councill 2001; Jifeng et al. 2005; Weishäupl 2002). Für eine erfolgreiche Softwareentwicklung ist die Entwicklung einer Softwarearchitektur ein wichtiger Faktor (Gharbi et al. 2015). Ein weiterer wichtiger Faktor ist die Verwendung von geprüfter Standardsoftware, wie z. B. Middleware (Bernstein 1996). Dieser Abschnitt liefert einen Überblick über die Begriffe Softwarekomponente, komponentenbasierte Architektur und komponentenbasierte Middleware.

2.1.1 Definition einer Softwarekomponente

In der Geschichte der Softwarekomponente wurden mehrere Definitionen zu deren Beschreibung erstellt. Eine allgemein akzeptierte Definition einer Softwarekomponente, welche auch in dieser Arbeit verwendet wird, hat Clemens Szyperski (Szyperski 2003) beschrieben.

„A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.“ (Szyperski 2003)

Eine solche Softwarekomponente, im Folgenden nur noch Komponente genannt, besitzt folgende Eigenschaften:

- Sie kommuniziert nur über klar definierte Schnittstellen mit anderen Komponenten.
- Anwendungsentwickler komponieren Komponenten nach deren Entwicklung zu Anwendungen.

Eine Komponente wird als geschlossene Einheit gesehen. Der innere Aufbau einer Komponente wird dabei nicht betrachtet. Einzig die Schnittstellen, über die die Komponente mit anderen kommuniziert, können betrachtet werden. Eine Komponente kann mehrere Schnittstellen besitzen, auf die andere Komponenten zugreifen können. Ebenso kann eine Komponente zum erfolgreichen Ausführen ihrer Aufgaben andere Komponenten benötigen, wie Abbildung 2-1 exemplarisch zeigt.

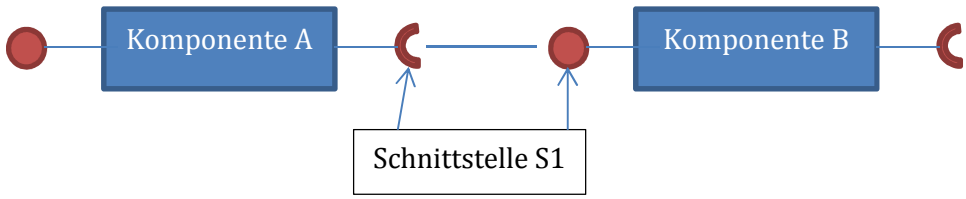

Abbildung 2-1: Darstellung zweier Komponenten

Das blaue Rechteck symbolisiert die Komponente. Der rote Kreis an der linken Seite der Komponente stellt einen Dienst über eine definierte Schnittstelle dar, den andere Komponenten nutzen können. Der rote Halbkreis auf der rechten Seite der Komponente beschreibt wiederum einen Dienst über eine Schnittstelle zu einer anderen Komponente, welche zum erfolgreichen Ausführen des Dienstes der Komponente benötigt wird. Damit zwei Komponenten zusammengeschaltet werden können, müssen beide die gleiche Schnittstelle verwenden.

Anwendungsentwickler haben im Vorfeld die Schnittstellen der Komponenten spezifiziert oder Schnittstellen von schon bestehenden Komponenten, welche verwendet werden sollen, ausgewählt. Nach Entwicklung der verbleibenden Komponenten werden diese zusammen mit den schon bestehenden Komponenten vom Anwendungsentwickler zu einer Anwendung verknüpft.

Aus den Eigenschaften von Komponenten ergeben sich für die Entwicklung solcher komponentenbasierter Anwendungen mehrere Vorteile:

- Komponenten können wiederverwertet werden.
- Komponenten können in einer Anwendung leicht ausgetauscht werden.
- Komponenten einer Anwendung können unabhängig voneinander entwickelt werden.

Durch die Verknüpfung von Komponenten über ausschließlich spezifizierte Schnittstellen können Komponenten ohne größeren Aufwand in weiteren Anwendungen wiederverwendet werden, welche dieselbe Schnittstelle spezifizieren. Dabei kann der Entwicklungsaufwand deutlich reduziert werden.

Außerdem lassen Komponenten sich einfacher ersetzen, z. B. wenn ein angeschlossener Sensor defekt ist oder ein neuer Sensor genutzt werden soll. Wenn die neue Komponente die gleichen Schnittstellen besitzt, kann die Komponente ohne großen Aufwand direkt ausgetauscht werden.

Der größte Vorteil liegt aber in der Tatsache, dass die Komponenten nach Spezifikation der verwendeten Schnittstellen unabhängig voneinander entwickelt werden können. Jede Gruppe kann sich dabei auf die einzelne Komponente konzentrieren, ohne die gesamte Anwendung kennen zu müssen. Dadurch kann die Entwicklung der einzelnen Komponenten parallel erfolgen, was wiederum eine Menge Entwicklungszeit spart. Besonders große Systeme und Anwendungen sind durch diese Eigenschaft überhaupt erst in akzeptabler Zeit umsetzbar.

Des Weiteren können Komponenten als Service entwickelt werden. Dabei werden Komponenten als Standard unabhängig zu einer Anwendung entwickelt. Diese Komponenten können dann direkt in Anwendungen verwendet werden, ohne sie ggf. von einer bestehenden Anwendung auf eine neue Anwendung anpassen zu müssen (Turner et al. 2003).

2.1.2 Softwarearchitektur

Jedes Softwaresystem besitzt eine Architektur, auch wenn sie nicht immer explizit vorher entwickelt wurde. Wie schon im letzten Kapitel beschrieben, ist eine gute Softwarearchitektur für den Erfolg einer Softwareentwicklung ein wichtiger Faktor.

Der Aufbau einer Softwarearchitektur ist an das zu entwickelnde Softwaresystem angelehnt. Daher wurde versucht, Softwaresysteme zu kategorisieren, für die eine Reihe von Architekturmustern existiert. Die Kategorisierung aus (Gharbi et al. 2015) teilt Softwaresysteme in drei Kategorien ein, eingebettete Systeme, Informationssysteme und mobile Systeme.

Zu Informationssystemen gehören Softwaresysteme, welche eine große Datenmenge an Informationen verwalten. Dabei muss ggf. eine große Menge an Nutzern gleichzeitig auf diese Daten zugreifen können. Beispiele dafür sind SAP-Systeme oder komplexe Simulationssysteme zur Wettervorhersage.

Bei eingebetteten Systemen existiert Software in physikalischen Gegenständen, wie z. B. Waschmaschinen oder Fertigungsanlagen. Unter Ressourcenbeschränkung der Hardware muss diese Software Funktionalitäten, wie z. B. Regelungs- oder Steuerungsfunktionen, mit hohen Ansprüchen an die Qualität der Ergebnisse ausführen können.

Mobile Systeme hingegen sind autonome oder semiautonome Einheiten, welche meist mit anderen Einheiten interagieren müssen, um ihre Aufgaben erfüllen oder einen Dienst für andere Einheiten anbieten zu können. Aufgrund ihrer Mobilität kann eine kontinuierliche Interaktion aber nicht garantiert werden. Es muss zu jedem Zeitpunkt mit Verbindungsabbrüchen gerechnet werden. Beispiele hierfür sind Smartphones oder Transportroboter.

Allerdings existieren Softwaresysteme, welche nicht eindeutig kategorisiert werden können, zum Beispiel wenn ein SAP-System über ein Smartphone verwendet wird. Zum einen enthält dieses System ein Informationssystem. Zum anderen werden mobile Systeme wie das Smartphone verwendet. Damit besitzt das System die Anforderung, eine große Datenmenge zu verwalten, und ebenfalls die Anforderung, mit Verbindungsabbrüchen bei Smartphones umgehen zu können. Aber meist besitzt jedes System einen Schwerpunkt, der eine Zuordnung zu einer der drei Kategorien ermöglicht.

Jede dieser Kategorien besitzt bestimmte Grundsystematiken bezüglich der Softwarearchitektur. Informationssysteme verfügen meist über eine Schichtenarchitektur, eingebettete Systeme bestehen überwiegend aus einzelnen Modulen, welche sich z. B. über eine Bus-Kommunikation austauschen, und mobile Systeme benötigen in ihrer Architektur die Möglichkeit, in einer dynamischen Umgebung zu arbeiten, um auf Verbindungsabbrüche reagieren zu können (Gharbi et al. 2015).

Auch besitzt jede Kategorie eine Menge von Entwurfsherausforderungen bzgl. der Architektur. Bei Informationssystemen steht die Frage nach der Datenhaltung im Vordergrund, wohingegen bei eingebetteten Systemen eher über das Scheduling aktiver Prozesse nachgedacht wird. Bei mobilen Systemen ist die sichere Datenübertragung ein wichtiger Punkt.

Eine Softwarearchitektur besitzt neben dem Aufbau der Software noch grundlegende Prinzipien und Regeln für die Organisation eines Systems. Dadurch werden Richtlinien für den gesamten Lebenszyklus, von der Entwicklung bis hin zur Wartung des Systems, beschrieben. Dazu muss auch die Umgebung, in der die Softwarearchitektur erstellt werden soll, mit betrachtet werden. Die Umgebung kann in vier Bereiche eingeteilt werden („Projektmanagement und Projektumfeld“, „Produktmanagement und Requirements Engineering“, „Ausführungsplattform und Betrieb“ und „Werkzeugumgebung und Entwicklung“) (Gharbi et al. 2015).

Meist existieren neben diesem Projekt noch weitere Projekte, welche mit dem System dieses Projekts interagieren sollen. Diese müssen bei der Entwicklung und Pflege der Softwarearchitektur betrachtet werden, da sich die Anforderungen dieser Projekte ändern können. Damit würden sich auch die Schnittstellen zu dem System ändern, für das eine Softwarearchitektur entwickelt werden soll.

Im Bereich Produktmanagement und Requirements Engineering werden die funktionalen und nicht funktionalen Anforderungen für das System bereitgestellt. Diese können sich über die Laufzeit der Softwareentwicklung ändern, weshalb sich dementsprechend auch die Softwarearchitektur ändern muss.

Meist ist für ein zu entwickelndes System eine Ausführungsplattform vorgegeben. Die Softwarearchitektur muss für diese Plattform entwickelt werden. Zusätzlich können aber auch Anforderungen an die Plattform selbst gestellt werden. Daher kann die Architektur auch eine Anpassung der Ausführungsplattform anstoßen.

Auch vorgegebene Entwicklungswerkzeuge oder Frameworks haben Einfluss auf die Struktur der Softwarearchitektur. Umgekehrt kann die Architektur auch die Auswahl der möglichen Frameworks definieren. Ein Beispiel für eine komponentenbasierte Architektur ist eine komponentenbasierte Middleware, welche neben Diensten für das zu entwickelnde Softwaresystem auch ein Komponentenmodell mitliefert (Bernstein 1996; Rütschlin et al. 2001).

Eine Softwarearchitektur kann also nicht isoliert entwickelt werden. Vielmehr benötigt sie Input aus diesen vier Bereichen.

Damit das zu entwickelnde Softwaresystem mit der Softwarearchitektur auch im Sinne des magischen Vierecks erfolgreich ist, müssen ebenfalls die Qualitätsziele in der Softwarearchitektur berücksichtigt werden. Die Qualitätsziele werden überwiegend aus den nicht funktionalen Anforderungen abgeleitet. Damit gehören sie zu dem konkret zu entwickelnden Softwaresystem. Die Qualitätsziele lassen sich fünf Qualitätsmerkmalen zuordnen (Funktionalität, Benutzbarkeit, Zuverlässigkeit, Effizienz und Änderbarkeit) (Gharbi et al. 2015). Mittels einer Auswahl der für das Projekt wichtigen Qualitätsziele kann anschließend das fertige Softwaresystem bewertet werden.

2.1.3 Komponentenbasierte Middleware

Eine Middleware ist ein Unterstützungssystem, welches sich zwischen einer Anwendung und dem Betriebssystem befindet (Bernstein 1996). Je nach Aufgabe der Middleware bietet diese den Anwendungen eine Reihe von Diensten an. Diese Dienste sind nicht für eine konkrete Anwendung erstellt.

Überwiegend wird von den existierenden Middleware-Lösungen verteiltes Arbeiten unterstützt. Das heißt, dass die Kommunikation zwischen den Komponenten einer Anwendung über Rech-

nergrenzen von der Middleware realisiert wird. Anwendungsentwickler können sich dabei hauptsächlich auf die eigentliche Anwendung konzentrieren. Ein weiterer Vorteil einer Middleware besteht in der Gewährleistung einer erwarteten Qualität. Eine Middleware wurde meist schon von anderen Anwendungsentwicklern und auch von den Middleware-Entwicklern auf Qualitätsziele hin geprüft. Daher kann anhand dieser Ergebnisse eine Middleware ausgewählt werden, welche die nötigen Qualitätsziele der eigenen Anwendung erfüllt.

Mittlerweile existiert eine Reihe von Middleware, welche die Kommunikation über Rechnergrenzen unterstützt. In Java gibt es das Kommunikationsprotokoll „Remote Method Invocation“ (RMI), über welches Instanzen von Klassen über Schnittstellen auf andere Klassen über das Netzwerk zugreifen können (Downing 1998). Das Framework CORBA besitzt, analog zum RMI-Protokoll, ebenfalls ein Kommunikationsprotokoll für den verteilten Zugriff auf Komponenten über Schnittstellen. Durch eine eigene Skriptsprache für diese Schnittstellen können sogar Komponenten unterschiedlicher Programmiersprachen miteinander kommunizieren (Siegel 2000). Über mitgelieferte Generatoren wird das Schnittstellenskript für die jeweilige Programmiersprache umgewandelt.

Eine komponentenbasierte Middleware bietet zusätzlich noch ein Komponentenmodell. Ein Komponentenmodell beschreibt zum einen die Struktur einer Komponente. Häufig wird dafür ein Metamodell verwendet. Über diese Struktur besitzen alle Komponenten einen einheitlichen Aufbau. Außerdem wird je nach Komponentenmodell eine Reihe von Eigenschaften der Komponente gegeben, wie z. B., dass eine Komponente mehrere Konfigurationen besitzt (Klus et al. 2007). Zum anderen beschreibt ein Komponentenmodell auch die Interaktion zwischen den Komponenten. Anwendungen, welche das Komponentenmodell einer Middleware verwenden, besitzen nun eine gemeinsame Basis, um einfacher miteinander zu kommunizieren (Rütschlin et al. 2001). In den letzten Jahren wurde mehrfach Middleware mit eigenen Komponentenmodellen entwickelt. Nachfolgend werden einige Beispiele von komponentenbasierter Middleware kurz vorgestellt.

Eines ist das Framework „Enterprise JavaBeans“ (EJB), welches vorgefertigte Komponentenmodelle für Komponenten mit unterschiedlichen Schwerpunkten besitzt. Das Modell „Entity Bean“ ermöglicht z. B. die Erstellung einer Datenbankkomponente für eine persistente Datenhaltung in Kombination mit einer Datenbank (Backschat und Rücker 2007).

Zusätzlich besitzt auch CORBA mit dem „CORBA Component Modell“ (CCM) ein Komponentenmodell. Die Schnittstellen zwischen den Komponenten werden dabei über sogenannte Ports realisiert. Je nach Auswahl eines Ports kann z. B. entschieden werden, ob zwei Komponenten miteinander über synchrone oder asynchrone Aufrufe kommunizieren (Wang et al. 2001).

Die Middleware RUNES ermöglicht mit ihrem Komponentenmodell das dynamische Starten und Beenden von Komponenten zur Laufzeit. Diese Middleware ist aus dem von der EU geförderten Projekt RUNES (Reconfigurable Ubiquitous Networked Embedded Systems) entstanden (Costa et al. 2005). Zu dem Komponentenmodell wurde für mehrere eingebettete Zielsysteme jeweils eine Implementierung erzeugt. Dadurch können Komponenten dieses Komponentenmodells auf unterschiedlichen Systemen laufen und trotzdem miteinander kommunizieren.

Die Middleware DREAM besitzt ebenfalls ein Komponentenmodell für eine dynamische Verschaltung von Komponenten. Durch die Unterteilung in primitive und zusammengesetzte Komponenten wird sogar eine hierarchische Verschaltung unterstützt. Allerdings besteht die Kom-

munikation zwischen den Komponenten nur aus dem Austausch rein asynchroner Nachrichten (Leclercq et al. 2005).

Bei der Middleware DAiSI können Komponenten in unterschiedlichen Konfigurationen arbeiten, wobei immer maximal eine davon abhängig von den verbundenen benötigten Komponenten ausgewählt ist. Je nach Konfiguration kann eine Komponente unterschiedliche Aufgaben ausführen oder eine Aufgabe mit unterschiedlichem Aufwand oder Qualität des Ergebnisses bearbeiten. Die Dienste, welche solch eine Komponente anbieten, sind ebenfalls von der ausgewählten Konfiguration abhängig. Wie auch bei der Middleware RUNES und DREAM werden die Verschaltung der Komponenten und die Auswahl der passenden Konfiguration pro Komponente zur Laufzeit bestimmt (Klus et al. 2007).

2.2 Selbstorganisierende Softwaresysteme

Selbstorganisierende Softwaresysteme können sich selbstständig an Änderungen der Umgebung anpassen. Die Änderungen beziehen sich auf das Verhalten des Systems über den Austausch ausgefallener Komponenten bis hin zur Anpassung der Struktur des Systems. Das ermöglicht mehr Flexibilität und verringert die Möglichkeit des Ausfalls des Systems. Wenn z. B. eine wichtige Einheit eines selbstorganisierenden Systems ausfällt, kann das System selbstständig Ersatz suchen oder sogar durch Starten dieser Einheit selbst für Ersatz sorgen.

Um dies zu gewährleisten, müssen zusätzliche Herausforderungen gemeistert werden (Lemos et al. 2013). Dies beginnt schon bei der Anforderungserhebung. In dieser Phase muss festgelegt werden, auf welche Art von Änderungen das selbstorganisierende System in welcher Weise reagieren soll. Anschließend müssen sich diese Anforderungen auch in der Softwarearchitektur des Systems wiederfinden lassen. Zum Beispiel für den dynamischen Austausch einer Komponente muss die Architektur das Einbinden und Auslösen dieser Komponente zur Laufzeit gewährleisten können. Des Weiteren müssen Prozesse entwickelt werden, welche die Notwendigkeit eines Austauschs erkennen und die Komponente anschließend austauschen können. Abschließend werden noch Verfahren benötigt, um den Erfolg der Komponentenauswechslung und die weiterhin korrekte Arbeitsweise des Systems verifizieren und validieren zu können (Lemos et al. 2013).

2.2.1 Die MAPE-K-Schleife

Um das zu gewährleisten, besitzt das System einen Organisationsprozess, die sogenannte MAPE-K-Schleife, welche bei selbstorganisierenden Systemen ein bewährtes Prinzip darstellt (Arcaini et al. 2015). Sie besteht aus vier Arbeitsschritten: Monitor, Analyse, Plan und Execute, welche auf einer gemeinsamen Wissensbasis (Knowledge) arbeiten.

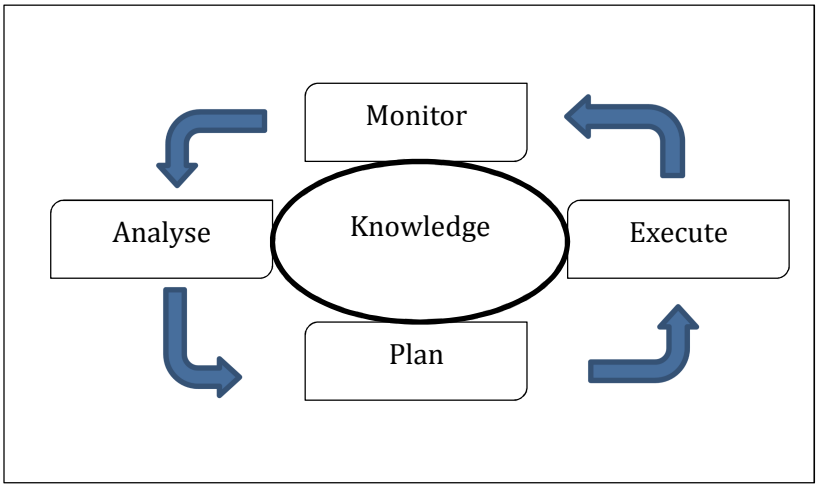

Abbildung 2-2: Darstellung des Organisationsprozesses für selbstorganisierende Softwaresysteme

Im Arbeitsschritt „Monitor“ prüft das System, ob im System oder in der Umgebung eine Änderung aufgetreten ist, die ein Handeln des Systems erforderlich macht. Im Abschnitt „Analyse“ wird ein beobachtetes Problem untersucht und für den nachfolgenden Arbeitsschritt kategorisiert. Im Schritt „Plan“ werden zu den analysierten Problemen passende Lösungen ermittelt und jeweils die beste Lösung wird ausgewählt. Mit diesen Lösungen wird zur Eliminierung der Probleme in Abschnitt „Execute“ das System neu organisiert. Anschließend wird wieder der Arbeitsschritt „Monitor“ ausgeführt (Di Nitto et al. 2008; Oreizy et al. 1999).

Selbstorganisierende Softwaresysteme werden ebenfalls im Bereich der komponentenbasierten Softwareentwicklung entwickelt. Dort entspricht eine Neuorganisierung des Systems meist einer Verschaltung einer Menge von Komponenten in dem System (Klus et al. 2007). Multiagentensysteme sind ein weiterer Bereich, in dem Selbstorganisation in Form von autonomen Agenten betrachtet wird.

2.2.2 Multiagentensysteme

Multiagentensysteme sind Systeme, welche aus selbstständig agierenden Einheiten, auch Agenten genannt, bestehen und zusammen ein gemeinsames Ziel anstreben. Eine Beispielfamilie biologischer Multiagentensysteme sind Ameisenkolonien. Jede Ameise besitzt dabei spezifische Aufgaben, welche sie selbstständig erfüllt (Futtersuche, Kolonieverteidigung, …). Aus manchen dieser Abläufe wurden Algorithmen entwickelt, welche als heuristische Lösungsverfahren für komplexe Optimierungsaufgaben verwendet werden (Dorigo und Stützle 2009). Solche Systeme werden seit ca. 1980 intensiv erforscht (Wooldridge 2009). Ein Agent eines Multiagentensystems besitzt zwei Fähigkeiten. Zum einen ist ein Agent eine atomare selbstagierende Einheit. Das heißt, jeder Agent entscheidet selbst, was er braucht und wie er agiert, um sein gegebenes Ziel zu erreichen. Zum anderen besitzt ein Agent die Fähigkeit, mit anderen Agenten zum Erreichen des Ziels zu interagieren. Aber anders als bei Services oder Komponenten besteht die Kommunikation in einem Multiagentensystem nicht in einer simplen Nutzung eines anderen Agenten. Wie Individuen müssen Agenten untereinander kooperieren, sich koordinieren oder miteinander verhandeln, um ihr Ziel zu erreichen. Dafür besitzt ein Agent eine Menge von Strategien, aus denen er jeweils die auswählt, welche ein optimales Erreichen seines Ziels verspricht. Die Auswahl der geeigneten Strategie kann außerdem von Regeln des Multiagentensystems beeinflusst werden. In einem kollaborierenden Multiagentensystem mit einem globalen Ziel z. B. sollte eine Strategie benötigt und gewählt werden, welche ein Erreichen des globalen Ziels ermöglicht (Wooldridge 2009; Jensen et al. 2016; Dennisen und Müller 2015).

2.3 Spieltheorie

In der Spieltheorie werden Entscheidungssituationen modelliert. Aber anders als in der klassischen Entscheidungstheorie ist der Erfolg des Einzelnen nicht nur durch sein eigenes Handeln, sondern auch von Aktionen anderer abhängig (Neumann und Morgenstern 2007; Russell und Norvig 2012; Wooldridge 2009). John von Neumann legte im Jahr 1928 die Grundlage der modernen Spieltheorie (Neumann 1928). 1944 veröffentlichte er zusammen mit Oskar Morgenstern das Buch „Theory of Games and Economic“ (Neumann und Morgenstern 2007), welches noch heute als wegweisender Meilenstein gilt. 1950 wurde von John F. Nash das sogenannte Nash-Gleichgewicht, als allgemeinere Lösungsmöglichkeit, entwickelt (Nash und others 1950). Das Nash-Gleichgewicht bestimmt eine optimale Entscheidung aller Mitspieler in einem Spiel. Das heißt, wenn alle Spieler ihre jeweilige Entscheidung aus dem Nash-Gleichgewicht wählen und ein beliebiger Spieler aus seinen Möglichkeiten eine andere Entscheidung wählt, wird er sich in dem Spiel nicht verbessert haben. Allerdings ist es möglich, dass mehr als ein Nash-Gleichgewicht bei einem Spiel existiert. Das Nash-Gleichgewicht gilt in dem Bereich als eines der bekanntesten Lösungskonzepte.

Die Spieltheorie findet in mehreren Bereichen Anwendung – darunter auch im Operations Research, in den Wirtschaftswissenschaften, in der Psychologie und auch in der Informatik. In der Informatik wurden daraus für unterschiedliche Anwendungsgebiete unterschiedliche Verfahren für die Spieler entwickelt, welche sich unter anderem mit der optimalen Verteilung von Ressourcen beschäftigen. Da die Spieler auch als unabhängige Agenten betrachtet werden können, wurde dieser Bereich ebenfalls im Bereich der Multiagentensysteme intensiv erforscht. Im Folgenden wird eine Auswahl der entwickelten Verfahren kurz vorgestellt, welche anschließend in dieser Arbeit verwendet werden.

2.3.1 Mechanismenentwurf

Der Mechanismenentwurf ermöglicht die Entwicklung eines Verfahrens, in dem der Einsatz mehrerer rationaler Agenten, welche ihr Ziel zu erreichen versuchen, eine Maximierung der globalen Nutzenfunktion zur Folge haben soll. Ein Mechanismus besteht aus einer Beschreibungssprache für die Strategien der Agenten. Zusätzlich existiert ein spezieller Agent, welcher die übrigen Agenten anhand ihrer Handlungen für die für eine spezifische Umgebung erstellten Ereignisregeln „belohnt“. Diese Ereignisregeln entscheiden dabei über Erfolg oder Misserfolg des Mechanismus (Russell und Norvig 2012).

2.3.2 Feilschen um Ressourcen

Beim einfachen Feilschen zweier Agenten besitzt einer der beiden Agenten eine Ressource, welche der andere Agent erwerben möchte. Beide Agenten nennen einen Preis als Gegenwert. Anschließend nähern sich beide Agenten in nachfolgenden Geboten mit ihrem jeweiligen Preis einander an, bis sie sich auf einen Preis geeinigt haben. Im Bereich der Verteilung von Ressourcen unter mehreren Agenten existiert eine Bewertungsfunktion, welche die möglichen Verteilungen untereinander hinsichtlich ihres Nutzens bewerten kann. Jeder Agent versucht dabei, das Beste für sich herauszuholen (Wooldridge 2009; Russell und Norvig 2012).

2.3.3 Auktion

Bei einer Auktion bietet ein Agent, der Auktionator, eine Ressource an und eine Menge von weiteren Agenten, die Bieter, gibt Gebote an den Auktionator. Der gibt wiederum dem Bieter mit dem höchsten Gebot den Zuschlag für die Ressource. Es existiert eine Vielzahl unterschiedlicher Varianten an Auktionen (Wooldridge 2009). Die Gebote sind in manchen Varianten sichtbar, in

anderen geheim. Dann gibt es Varianten, wo jeder Bieter nur ein Gebot abgeben kann, und Varianten, wo jeder Bieter innerhalb einer Zeitspanne mehrere Gebote abgeben und dabei die Auktion beobachten kann. Auch die Ermittlung des zu zahlenden Preises des Höchstbieters ist unterschiedlich. Z. B. wird bei manchen Varianten anstatt des Gebots des Höchstbieters das Gebot des zweithöchsten Bieters vom Höchstbieter gezahlt.

3 Problembeschreibung

Im vorherigen Kapitel wurden Grundlagen zur komponentenbasierten Softwareentwicklung und dynamisch adaptive Infrastrukturen beschrieben. Das Ziel dieses Kapitels besteht darin, anhand eines Anwendungsbeispiels eine Problemstellung aus diesem Bereich abzuleiten und aufzuzeigen, dass dieses Problem aktuell noch nicht ausreichend erforscht ist.

In dem Anwendungsbeispiel werden Komponenten zur Laufzeit zu Anwendungen verschaltet. Aus diesem Grund wird zur Beschreibung dieser Komponenten das Komponentenmodell von Holger Klus verwendet, weil das Modell Komponenten und Anwendungen beschreibt, die zur Laufzeit dynamisch verbunden werden können (Klus 2013).

Dieses Kapitel beginnt mit einer Erläuterung des verwendeten Komponentenmodells von Holger Klus. Anschließend wird das Anwendungsbeispiel unter Verwendung des Modells beschrieben. Aus dem Anwendungsbeispiel wird die Problemstellung herausgearbeitet und verwandte Arbeiten zur Lösung des Problems werden betrachtet.

3.1 Grundlegender Rahmen komponentenbasierter dynamisch adaptiver Anwendungen

Dieses Unterkapitel beschäftigt sich mit der Beschreibung von Anwendungen und Komponenten mittels eines bestehenden Komponentenmodells. Unter Verwendung dieses Modells wird in den darauffolgenden Unterkapiteln das Anwendungsbeispiel beschrieben und die Problemstellung aus dem Anwendungsbeispiel abgeleitet.

3.1.1 Aufbau des Komponentenmodells

Da Teile des Komponentenmodells von Holger Klus zur Beschreibung der Problemstellung nicht benötigt werden, sind diese zum leichteren Verständnis aus dem Modell entfernt worden. Eine vollständige und ausführliche Beschreibung des Modells kann seiner Arbeit (Klus 2013) entnommen werden.

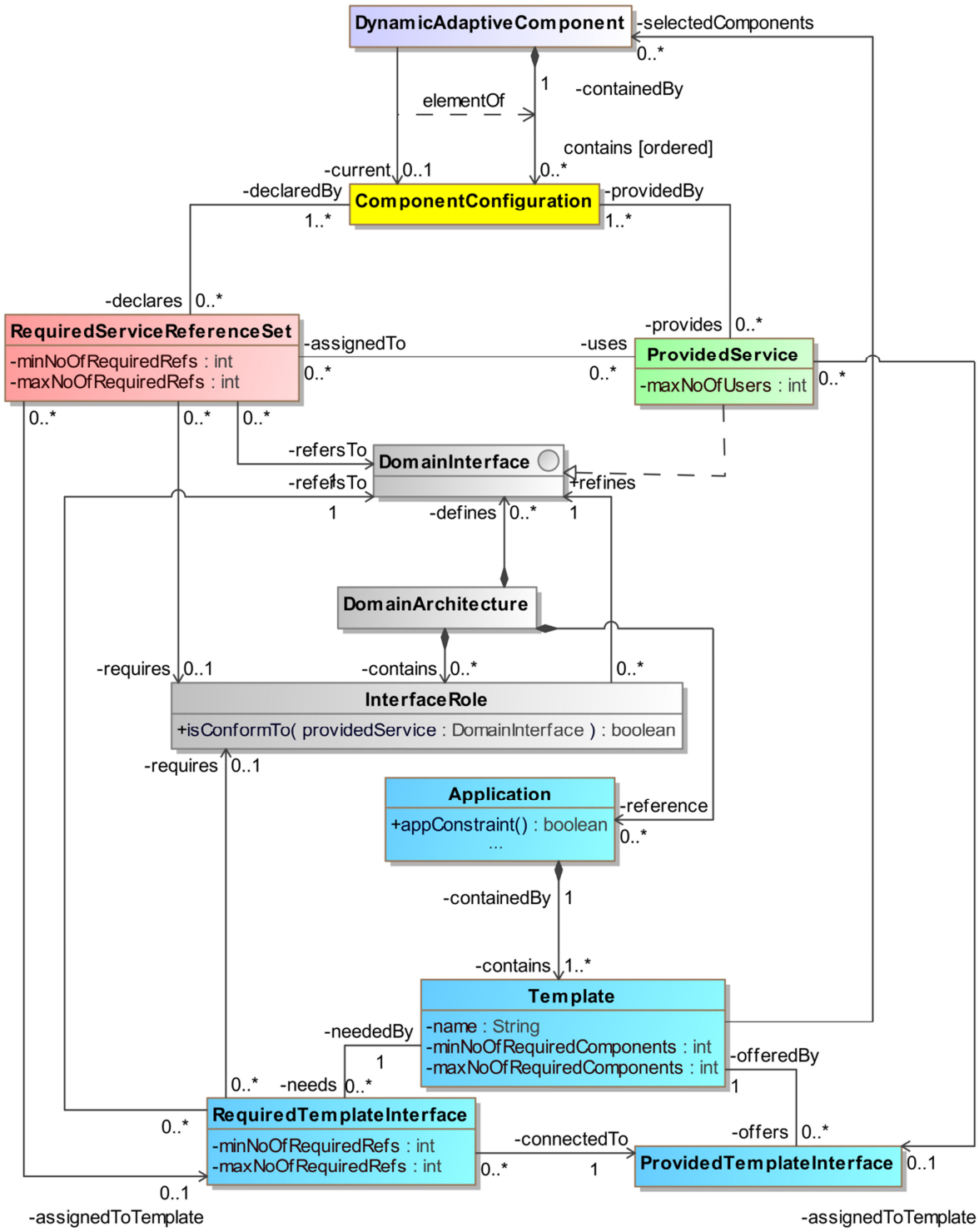

Abbildung 3-1: Komponentenmodell

Quelle: in Anlehnung an (Klus 2013), Seite 112

Abbildung 3-1 beschreibt den konzeptionellen Aufbau von Anwendungen und Komponenten. Eine **Komponente** (DynamicAdaptiveComponent) besitzt eine sortierte Menge von **Komponentenkonfigurationen** (ComponentConfiguration). Diese Konfigurationen unterscheiden sich untereinander in der Menge der **angebotenen Dienste** (ProvidedService) und in der Menge der **benötigten Dienste** (RequiredServiceReferenceSet). Jeder Dienst verweist auf eine **Domänen-**

schnittstelle (DomainInterface), über die benötigte und dazu passende angebotene Dienste identifiziert werden können.

Die angebotenen Dienste können nur mit einer maximalen Anzahl benötigtet Dienste verknüpft werden (maxNoOfUsers). Die benötigten Dienste müssen hingegen jeweils mit einer definierten Menge (minNoOfRequiredRefs-maxNoOfRequiredRefs) angebotener Dienste verknüpft sein.

Die Domänenschnittstellen sind einer **Anwendungsdomäne** (DomainArchitecture) zugeordnet, welche die Domäne definiert, in der die Komponentenkonfigurationen ihre Dienste anbieten. In der **Schnittstellenrolle** (InterfaceRole) kann auf die Methoden des angebotenen Dienstes und auf die damit zu erhaltenen Informationen der Komponente zugegriffen werden. So können von Komponentenkonfigurationen unterschiedlicher Komponenten, welche jeweils den gleichen Dienst mit der gleichen Domänenschnittstelle anbieten, für andere Komponenten, welche diesen Dienst benötigen, automatisch die Komponenten ausgewählt werden, welche die Bedingungen der zugehörigen Schnittstellenrolle erfüllen. Für die Problemstellung wird die Schnittstellenrolle allerdings nicht weiter betrachtet.

Anwendungen (Application) sind ebenfalls einer Anwendungsdomäne zugeordnet. Sie besitzen eine Menge von **Schablonen** (Template), welche bei der Verschaltung jeweils mit einer definierten Menge von Komponenten verknüpft werden müssen. Außerdem muss der Schablone eine Menge (minNoOfRequiredComponents-maxNoOfRequiredComponents) von Komponenten in der jeweiligen Komponentenkonfiguration zugeordnet werden, welche den Anforderungen der Schablone entsprechen. Analog zu Komponentenkonfigurationen besitzen Schablonen eine Menge von **anbietenden Schnittstellenschablonen** (ProvidedTemplateInterface) und eine Menge von **benötigten Schnittstellenschablonen** (RequiredTemplateInterface). Die benötigten Schnittstellenschablonen besitzen, wie die benötigten Dienste der Konfigurationen, ebenfalls eine definierte Menge (minNoOfRequiredRefs-maxNoOfRequiredRefs) von möglichen Verbindungen. Eine passende Konfiguration einer Komponente muss demnach genau einen Dienst für jede anbietende Schnittstellenschablone anbieten und genau die Dienste passend zu den benötigten Schnittstellenschablonen der Schablone benötigen. Außerdem muss die definierte Menge der benötigten Dienste außerhalb der Menge der zugehörigen benötigten Schnittstellenschablonen liegen, damit die Konfiguration eine Komponente die Schablone vollständig unterstützen kann.

Allerdings lassen sich nicht alle Bedingungen allein über die Struktur der Anwendung abbilden. Zum Beispiel kann eine Bedingung mit der in Abbildung 3-3 beschriebenen Aussage, dass eine mit der Anwendung verknüpfte Komponente ausschließlich mit Komponenten dieser Anwendung verbunden sein soll, nicht über die Struktur ausgedrückt werden. Daher besitzt die Anwendung zusätzlich noch die Methode „appConstraint“, in der eine Menge von weiteren Bedingungen geprüft werden kann.

3.1.2 Grafische Notation

Zum besseren Verständnis werden die Konzepte mittels einer grafischen Notation näher erläutert. Die grafische Notation wird anschließend in den folgenden Kapiteln für die Beschreibung der Komponenten und Anwendungen des Anwendungsbeispiels verwendet.

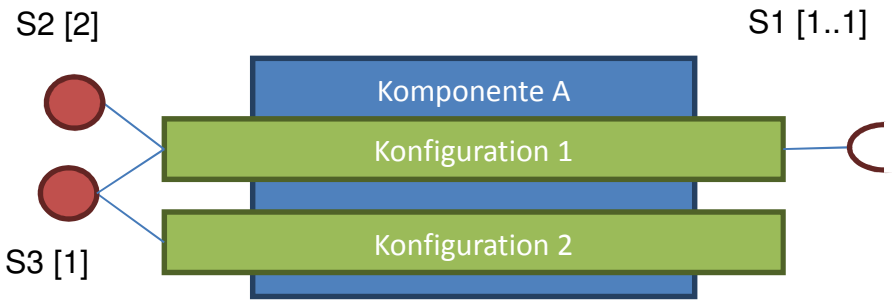

Abbildung 3-2: Grafische Notation einer Komponente

In Abbildung 3-2 wird die Komponente „Komponente A“ (DynamicAdaptiveComponent) mit zwei Komponentenkonfigurationen (ComponentConfiguration) dargestellt. „Konfiguration 2“ bietet einen Dienst (ProvidedService) über die Schnittstelle „S3“ (DomainInterface) für maximal einen benötigten Dienst an (maxNoOfUsers=1). Repräsentiert wird der angebotene Dienst über einen Kreis. „Konfiguration 1“ bietet den Dienst über die Schnittstelle „S3“ ebenfalls an. Außerdem bietet sie noch einen Dienst über die Schnittstelle „S2“ mit Raum für maximal zwei benötigte Dienste an. Zum Arbeiten benötigt „Konfiguration 1“ genau einen weiteren Dienst (RequiredServiceReferenceSet), welcher die Schnittstelle „S1“ realisiert. Die benötigten Dienste werden durch einen leeren Halbkreis dargestellt.

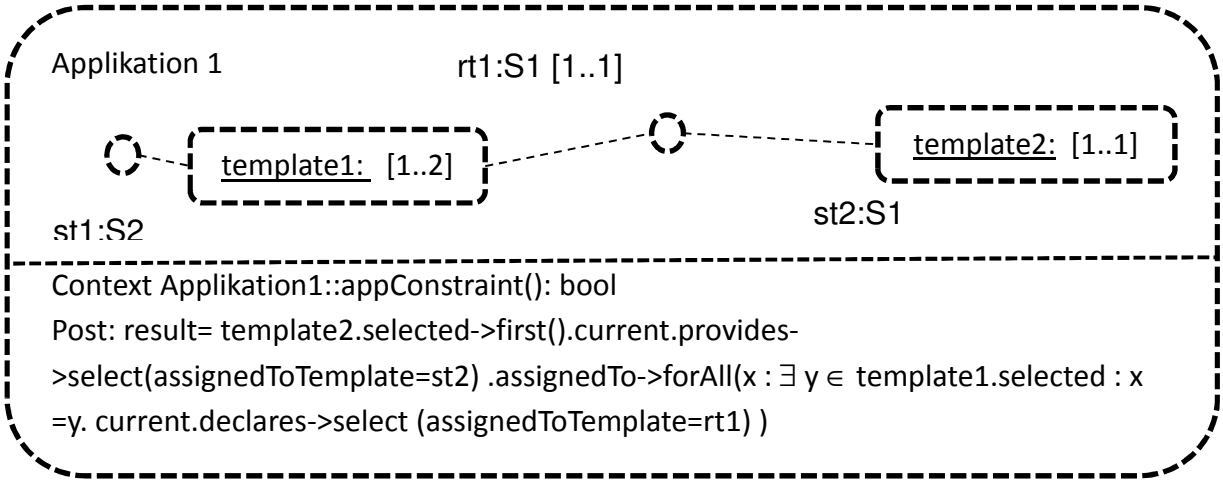

Abbildung 3-3: Grafische Notation einer Application

In Abbildung 3-3 wird die grafische Notation der Anwendungsbeschreibungen der Anwendung „Applikation 1“ gezeigt. Die Anwendungsbeschreibungen enthalten im oberen Bereich die Struktur der Anwendung (Application). Im unteren Bereich werden die Bedingungen der Anwendungskonfiguration mittels OCL beschrieben. Diese werden nach der Implementierung der Anwendung in der Methode „appConstraint“ geprüft.

Die in Abbildung 3-3 dargestellte Anwendung besitzt zwei Schablonen (Template). Die Schablone „template1“ benötigt ein bis zwei Komponenten in einer Komponentenkonfiguration, welche Dienste für die Schnittstellenschablonen st1 und rt1 besitzen, also jeweils einen Dienst über die Schnittstelle „S2“ anbieten und einen Dienst über die Schnittstelle „S1“ benötigen. Die Schablone „template2“ benötigt genau eine Komponente, welche mit ihrem angebotenen Dienst über die Schnittstelle „S1“ alle Komponenten in der Schablone „template1“ bedienen können muss (maxNoOfUsers), die jeweils den Dienst über die Schnittstelle „S1“ benötigen. Die Bedingung der Anwendung im unteren Bereich der grafischen Notation besagt, dass alle Verbindungen über die

Schnittstelle „st2“ mit dem Schnittstellennamen „S1“ von „template2“ nur mit Komponenten aufgebaut sein sollen, welche mit „template1“ verbunden sind.

3.1.3 Beschreibung des daraus resultierenden dynamischen Systems

Mit dem Komponentenmodell aus Abbildung 3-1 kann der Aufbau von Komponenten und Anwendungen beschrieben werden. In diesem Abschnitt werden nun die dynamischen Beziehungen zwischen Komponenten und Anwendungen, welche zur Laufzeit gesetzt werden, beschrieben. Auch wird das Modell dahin gehend erweitert, damit zum einen das System und zum anderen die Nutzer des Systems beschrieben werden können, siehe Abbildung 3-4. Anders als bei (Klus 2013) enthält das System neben einer Menge von Komponenten nun auch mehrere Anwendungen. Zusätzlich muss in diesem System verdeutlicht werden, dass eine Komponente oder eine Anwendung jederzeit von einem Nutzer gestartet werden kann. Dies verändert den Zustand des Systems bezogen auf die Anzahl an gestarteten Komponenten oder Anwendungen.

Komponenten und Anwendungen eines Systems, welche nach der Beschreibung aus Kapitel 3.1.1 erzeugt wurden, können anschließend von einer dynamisch adaptiven Middleware zur Laufzeit konfiguriert werden. Eine Konfiguration einer Komponente wird dabei von einer **Middleware** ausgewählt, wenn alle dafür benötigten Dienste mit angebotenen Diensten anderer Komponenten verbunden sind. Die ausgewählte Konfiguration einer Komponente wird in dem Attribut „current“ gespeichert. Wenn mehrere Komponentenkonfigurationen aus der Liste „contains“ potenziell ausgewählt werden können, wird beim Durchlauf der Liste die erste passende Komponentenkonfiguration gewählt. Im Folgenden werden nur Komponenten mit genau einer Komponentenkonfiguration betrachtet.

Über das Verknüpfen von benötigten Diensten mit angebotenen Diensten über die gleiche Schnittstelle wird bei einer Komponente die jeweilige Komponentenkonfiguration ausgewählt. Das Verknüpfen von benötigten Diensten einer Komponente mit angebotenen Diensten einer anderen Komponente über die gleichen Schnittstellen entspricht der Zuweisung des Attributs „uses“ der Klasse RequiredServiceReferenceSet. Dies wird im Folgenden **Verschaltung** zweier Komponenten genannt.

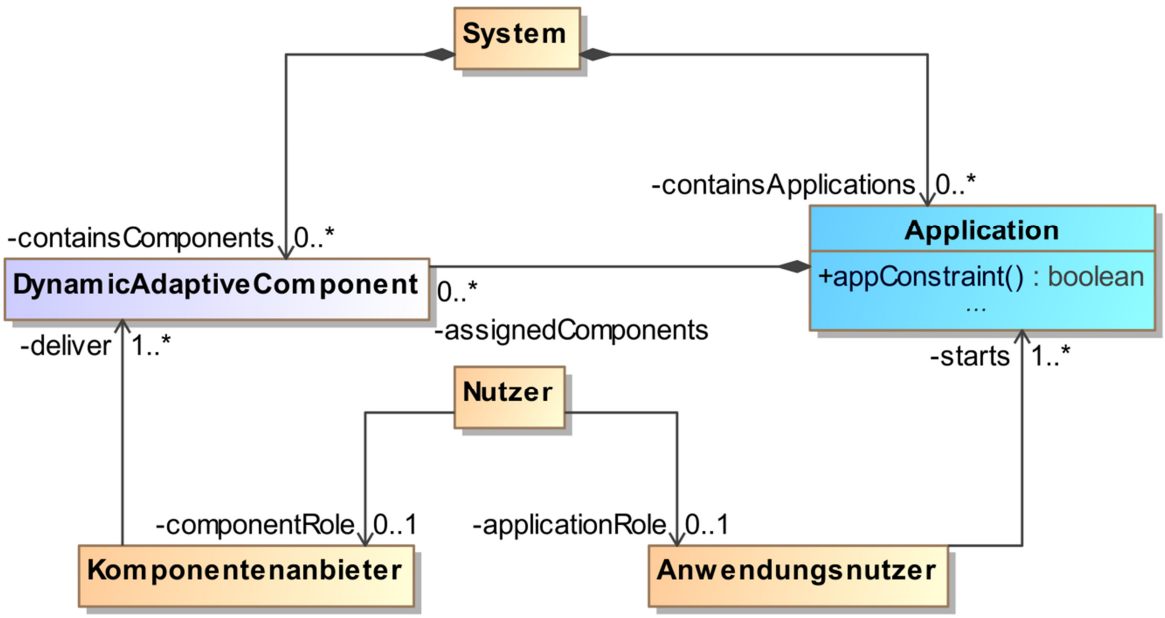

Abbildung 3-4: Systemmodell

Wie in Abbildung 3-4 dargestellt ist, enthält das zu betrachtende **System** alle von Nutzern gestarteten Komponenten und Anwendungen zur Konfiguration für die Middleware. Des Weiteren existieren noch Komponenten, welche von Anwendungen gestartet werden und diesen über „assignedComponents“ fest zugeordnet sind. Das betrifft z. B. Komponenten, welche Benutzerinteraktionen für eine Anwendung realisieren und daher exklusiv mit dieser Anwendung verbunden sind. Da diese Komponenten einer Anwendung fest zugeordnet sind, sind sie von der dynamischen Zuordnung zu einer Anwendung ausgenommen.

Ein **Nutzer** des Systems kann von einer Person oder einem externen System repräsentiert werden. Ein Nutzer kann zwei Rollen besitzen. Als **Komponentenanbieter** liefert der Nutzer mindestens eine Komponente in das System. Als **Anwendungsnutzer** kann ein Nutzer mindestens eine Anwendung im System starten. Die Rollen schließen sich nicht gegenseitig aus, weshalb ein Nutzer auch beide Rollen ausfüllen kann. Des Weiteren ändert sich über die Nutzer auch die Menge an Komponenten und Anwendungen im System. Auf diese Änderungen zur Laufzeit muss die Middleware entsprechend reagieren können.

Die Verschaltung einer Menge von Komponenten des Systems, welche einer Anwendung zugeordnet sind und alle Anforderungen dieser Anwendung erfüllen, erzeugt eine **Anwendungskonfiguration**. Die Anwendungskonfiguration wird über die Zuweisung der Komponenten zu den Schablonen über das Attribut „selectedComponents“ in Abbildung 3-1 dargestellt. Dabei werden auch die Attribute „assignedToTemplate“ mit den angebotenen und benötigten Diensten berücksichtigt. Alle Anwendungskonfigurationen eines Systems bilden wiederum zusammen die **Systemkonfiguration**. Diese kann sich zur Laufzeit dynamisch an die sich ändernde Anzahl von Anwendungen und Komponenten anpassen.

Eine **Anwendungsdomänenkonfiguration** hingegen bezeichnet eine Menge von Anwendungskonfigurationen, welche derselben Anwendungsdomäne zugeordnet sind. Ausschließlich Anwendungskonfigurationen einer Anwendungsdomänenkonfiguration können untereinander um Komponenten konkurrieren, weil die zugehörigen Schnittstellen nur in ihrer Anwendungsdomäne definiert sind.

3.1.4 Grafische Notation des Systems

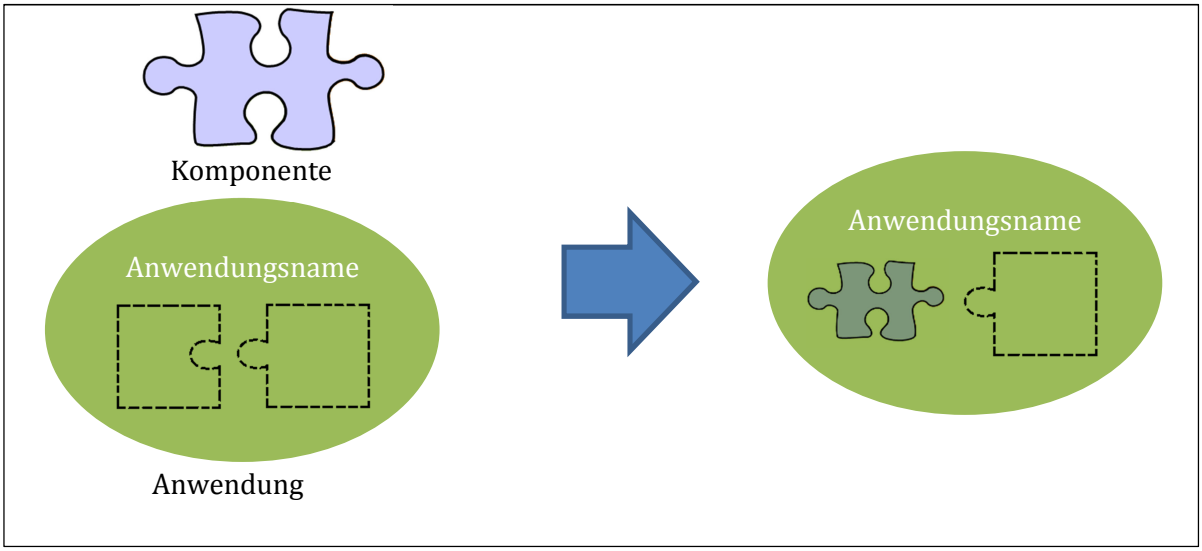

Abbildung 3-5: Grafische Notation des Systems

In Abbildung 3-5 werden zwei Systemkonfigurationen eines Systems grafisch dargestellt. Getrennt werden die beiden Systemkonfigurationen durch einen blauen Pfeil. Der Pfeil zeigt die erwartete Zielsystemkonfiguration, welche aus der bestehenden Systemkonfiguration erstellt werden soll.

Jede gestartete Komponente wird über ein grafisches Symbol repräsentiert. Jede gestartete Anwendung wird mittels einer grünen Ellipse dargestellt. Innerhalb der Ellipse werden der Name der Anwendung und jeweils eine Skizze für jedes grafische Symbol einer benötigten Komponente angezeigt. Jede Skizze entspricht dabei einer Schablone (Template) aus dem beschriebenen Komponentenmodell von Abbildung 3-1. Bei Zuordnung einer Komponente zu einer Anwendung wird die Skizze durch das grafische Symbol der Komponente ersetzt. Wenn eine Komponente mehreren Anwendungen zugeordnet ist, wird das grafisch durch die Überlappung der zugehörigen Ellipsen dargestellt. Die genaue Verschaltungsstruktur der Anwendungen wird zwecks einfacher Darstellung nicht abgebildet, sondern erst im nachfolgenden Kapitel mittels des Komponentenmodells aus Abbildung 3-1 gezeigt.

In den folgenden Kapiteln werden die Anwendungen und Komponenten des Anwendungsbeispiels mittels der grafischen Notation des Komponentenmodells beschrieben. Anschließend wird daraus das Problem herausgearbeitet.

3.2 Anwendungsbeispiel: Smart Airport

Aktuell werden Anwendungsbereiche für dynamisch adaptive Komponentenverschaltungssysteme intensiv erforscht – darunter fällt auch das Projekt IT-Ökosystem (Herold et al. 2008), bei dem anhand des Anwendungsbeispiels der Smart-City die Anforderungen in Bezug auf die Beherrschbarkeit eines sehr großen dynamischen Systems untersucht wurden. Dabei ist u. a. die Frage aufgetaucht, wie bei der Verschaltung von Komponenten Konflikte behandelt werden können. Ein Konflikt entsteht, wenn mehr Anwendungen einen Dienst nutzen möchten als die im System vorhandenen Komponenten diesen Dienst anbieten. Diese Arbeit beschäftigt sich mit dieser Frage und erarbeitet eine Antwort darauf.

Als Motivation dieser Frage wurde ein fiktives Anwendungsbeispiel für einen Smart-Airport erstellt. In dem Beispiel arbeitet eine begrenzte Anzahl von Robotern zum Transport von Gepäckwagen und es läuft eine bestimmte Anzahl von Anwendungen, welche in verschiedenen Szenarien um diese Roboter konkurrieren. Die Roboter können dabei entweder einen Gepäckwagen transportieren oder mit anderen Robotern zusammen eine Kolonne von Gepäckwagen befördern. Die Roboter dienen als begrenzte Ressource, da sich ein physikalischer Roboter nicht an zwei Orten zur gleichen Zeit befinden kann. Allerdings kann die Roboterkomponente prinzipiell mehrere Anwendungen, welche das gleiche Ziel besitzen, gleichzeitig bedienen. Um das Anwendungsbeispiel vorstellen zu können, müssen erst die darin verwendeten Anwendungen beschrieben werden, siehe Abbildung 3-6.

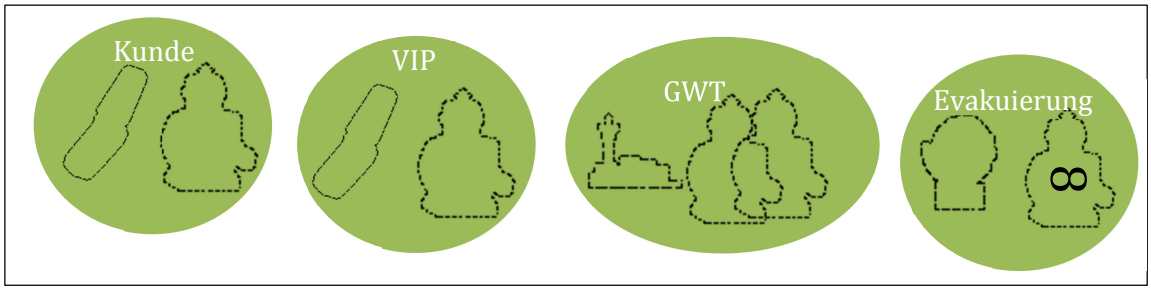

Abbildung 3-6: Liste der Anwendungen des Anwendungsbeispiels

Die erste Anwendung besitzt den Namen „Kunde“ und benutzt einen Roboter zum Transport des Gepäcks eines Kunden, welcher diese Anwendung ausführt, zu einem Zielort. Der Zielort muss bei Anwendungsstart noch vor der Suche nach einem Roboter ausgewählt werden. Durch diese Eigenschaft der Anwendung besteht die Möglichkeit, dass sich mehrere Anwendungen mit gleichem Ziel denselben Roboter teilen können.

Mittels der nächsten Anwendung „VIP“ wird ebenfalls ein Roboter für den Gepäcktransport eines Reisenden verwendet. Aber im Gegensatz zu der ersten Anwendung besitzt die Anwendung „VIP“ die Einschränkung, den Roboter exklusiv zu nutzen. Aus diesem Grund wird auch kein vorher gesetzter Zielort benötigt, welcher mit anderen Anwendungen bei der Robotersuche abgeglichen werden muss.

„Gepäckwagentransport“ (GWT) ist eine weitere Anwendung. Durch diese Anwendung kann der Flughafen den Transport einer Kolonne von Gepäckwagen zu einem Zielort organisieren. Zum Transport einer Kolonne von Gepäckwagen werden allerdings zwei Roboter benötigt. Ein Roboter übernimmt dabei vorne an der Kolonne die Aufgabe, das Ziel anzusteuern. Der zweite Roboter sorgt am Ende der Kolonne für den Antrieb. Die Anwendung erlaubt es anderen Anwendungen mit gleichem Ziel wie dieser, ihre Roboter ebenfalls nutzen zu können, z. B. der Anwendung „Kunde“. Aus diesem Grund wird bei der Anwendung „Gepäckwagentransport“ ebenfalls vor der Suche nach den Robotern ein Zielort definiert.

Die vierte Anwendung heißt „Evakuierung“ und ist eine Notfallanwendung. Wenn diese im Falle einer Evakuierung gestartet wird, sollen alle freien Roboter und alle gebundenen Roboter, welche aktuell von den anderen drei Anwendungen benutzt werden, mit dieser Notfallanwendung verknüpft werden. Das Ziel dieser Anwendung besteht darin, dafür zu sorgen, dass die Roboter den fliehenden Personen die Fluchtwege nicht versperren.

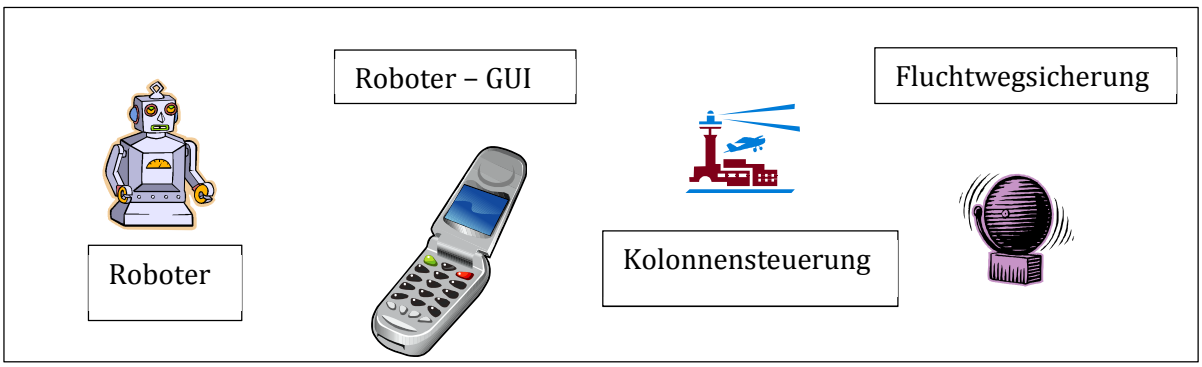

Abbildung 3-7: Liste der Komponenten des Anwendungsbeispiels

Den vier eben beschriebenen Anwendungen können vier Komponenten zugeordnet werden, siehe Abbildung 3-7. Die Komponente „Roboter“ leitet Steuerbefehle direkt an den angeschlossenen physikalischen Roboter weiter. Über die Komponente „Roboter – GUI“ sendet eine Person mittels einer grafischen Oberfläche Steuerbefehle an eine Roboterkomponente. Die Komponente „Kolonnensteuerung“ nutzt zwei Roboterkomponenten, um eine Gepäckwagenkolonne an eine definierte Position zu befördern. Die Komponente „Fluchtwegsicherung“ steuert alle Roboter über ihre jeweilige Roboterkomponente weg von den Fluchtwegen.

3.3 Beschreibung der Szenarien

Das zu beschreibende Anwendungsbeispiel besteht aus fünf Szenarien. In den Szenarien, welche jeweils ein System darstellen, werden die vier Anwendungen aus Kapitel 3.2 verwendet. Jedes der beschriebenen Szenarien liefert eine andere Anforderung, wie die in dem Anwendungsbeispiel enthaltenen Komponenten zu laufenden Anwendungen verschaltet werden sollen.

In den Szenarien werden die Komponenten „Roboter – GUI“, „Kolonnensteuerung“ und „Fluchtwegsicherung“ sofort und explizit einer gestarteten Anwendung zugeordnet. Damit konzentrieren sich die Szenarien auf die Zuordnung der Roboterkomponenten zu den gestarteten Anwendungen und den dabei auftretenden Konflikten. Nachfolgend werden die fünf Szenarien grafisch dargestellt und kurz beschrieben. Die Beschreibung beinhaltet auch die Anforderung für die optimale Systemkonfiguration in dem jeweiligen Szenario.

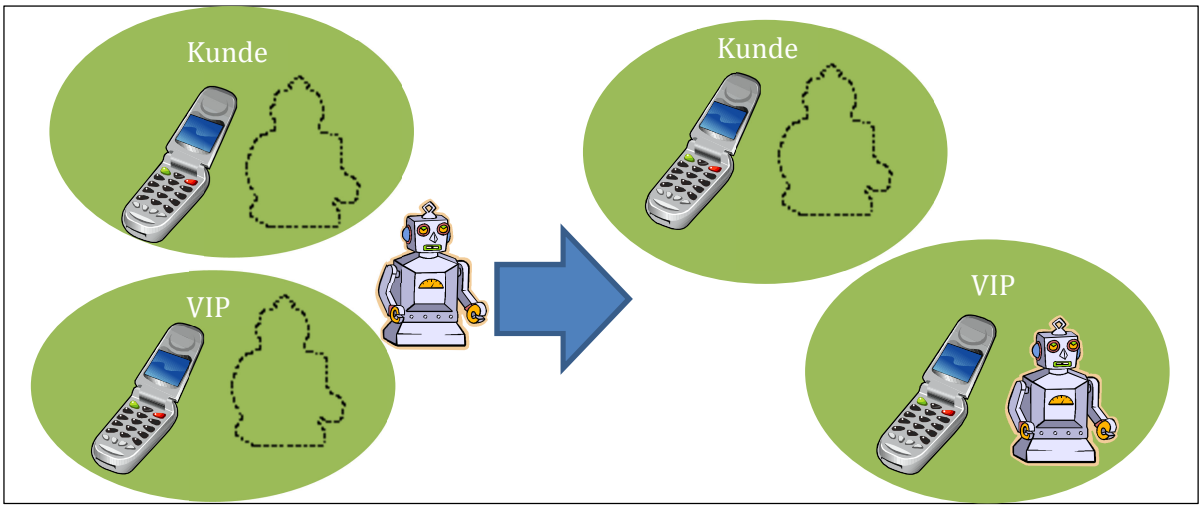

Abbildung 3-8: Szenario1: VIP hat vor einem Kunden Vorrang.

Abbildung 3-8 zeigt das 1. Szenario des Anwendungsbeispiels. In diesem Szenario existiert eine Roboterkomponente, jeweils eine Anwendung „VIP“ und „Kunde“ und zwei Komponenten „Roboter – GUI“. Die Anwendungen „VIP“ und „Kunde“ benötigen, wie in der Anwendungsbschreibung dargestellt, beide jeweils zum Ausführen ihrer Aufgaben noch eine Roboterkomponente. Da die VIP-Anwendung die Komponente nur exklusiv nutzen kann, gibt es in dem Szenario zwei Lösungsmöglichkeiten. Möglichkeit A lautet, dass die Kundenanwendung die Roboterkomponente erhält, und Möglichkeit B spricht der VIP-Anwendung die Roboterkomponente zu. Da der VIP einen höheren Status als der Kunde besitzen soll, wird demnach die Roboterkomponente der VIP-Anwendung zugewiesen.

In dem beschriebenen Szenario werden zwei Anwendungen gegenübergestellt, wobei eine Anwendung den Vorrang vor einer anderen Anwendung erhalten soll. Um dies entscheiden zu kön-

nen, ist eine Gewichtung aller Anwendungen notwendig. Dadurch entsteht eine Reihenfolge bei der Verschaltung von Anwendungen.

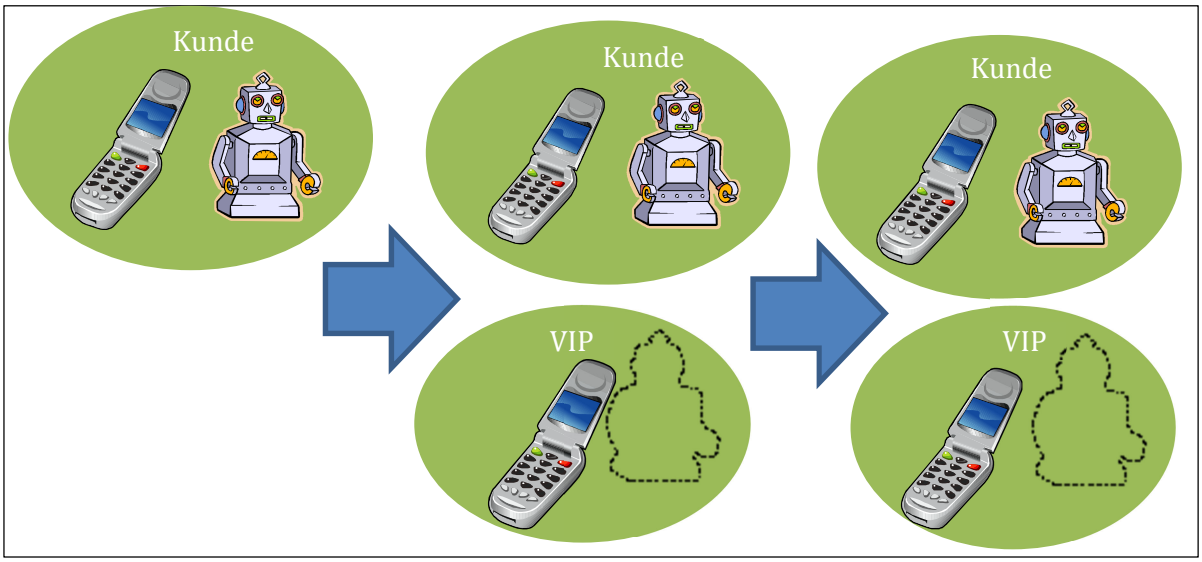

Abbildung 3-9: Szenario 2: VIP hat keinen Vorrang vor einem Kunden, welcher schon einen Roboter nutzt.

In Szenario 2 existieren die gleichen zwei Anwendungen „Kunde“ und „VIP“, zwei Komponenten „Roboter – GUI“ und eine Komponente „Roboter“, wie in Szenario 1. Aber da die VIP-Anwendung in diesem Szenario erst einige Zeit nach der Kundenanwendung gestartet wurde, hat sich die Roboterkomponente schon mit der Kundenanwendung verbunden. Obwohl die VIP-Anwendung, wie in Szenario 1 dargestellt, die höhere Gewichtung besitzt, soll sie trotzdem nicht fähig sein, die aktuelle Verbindung der Roboterkomponente mit der Kundenanwendung zu trennen. D. h., wie in Abbildung 3-9 gezeigt, die aktuelle Systemkonfiguration soll in diesem Szenario unverändert bleiben.

Bei dem 2. Szenario wird eine Anforderung für die Zuordnung einer Komponente zu einer Anwendung hinzugefügt. Nur freie Komponenten sollen demnach für eine Zuordnung zur Verfügung stehen. Es ist zwar denkbar, dass auch gebundene Komponenten, die evtl. noch weitere Kapazitäten besitzen, mit weiteren Anwendungen verknüpft werden können, wenn die bestehenden Verbindungen nicht beeinträchtigt werden. Aber dieser Aspekt verkompliziert die zu suchende Lösung und steuert keine weitere grundlegende Anforderung zu dem Problem bei, weshalb der Fall in dieser Arbeit nicht näher betrachtet wird. Daher wird sich die Zuordnung nur auf freie Komponenten beschränken. Eine Komponente wird in dieser Arbeit als frei bezeichnet, wenn sie keiner Anwendung zugeordnet ist. Anwendungen zugeordnete Komponenten werden als gebunden bezeichnet.

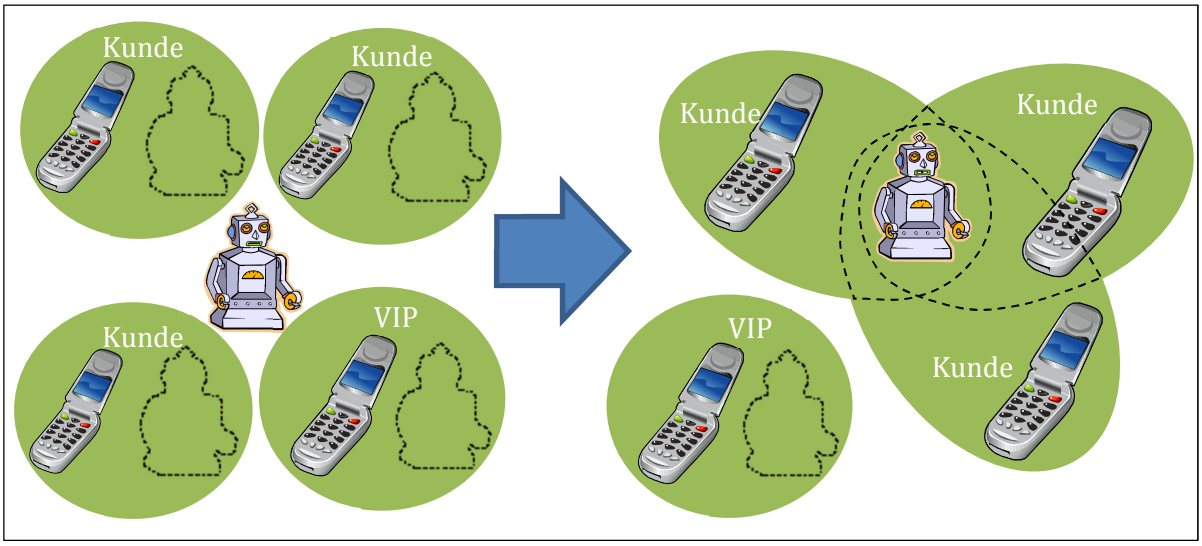

Abbildung 3-10: Szenario 3: Eine Gruppe von Kunden mit gleichem Ziel hat vor einem VIP Vorrang.

Szenario 3 besteht, wie in Abbildung 3-10 dargestellt ist, aus einer Anwendung „VIP“ und drei Anwendungen „Kunde“. Die Kundenanwendungen besitzen in diesem Szenario alle das gleiche Ziel. Außerdem existieren im System noch eine freie Roboterkomponente und vier Komponenten „Roboter – GUI“. In Szenario eins wurde gezeigt, dass eine VIP-Anwendung eine höhere Gewichtung als eine Kundenanwendung besitzt. Da sich hier aber drei Kundenanwendungen die Roboterkomponente aufgrund des gleichen Ziels teilen können, sollen die drei Anwendungen „Kunde“ vor der Anwendung „VIP“ mit der Roboterkomponente verbunden werden. Wie in Abbildung 3-10 illustriert, kann damit eine höhere Anzahl an Anwendungen laufen, was in diesem Szenario die optimale Systemkonfiguration darstellt.

Szenario 3 relativiert die Gewichtung aus Szenario 1, da hier eine Anwendungsdomänenkonfiguration von gleichen zusammenarbeitenden Anwendungen geringerer Gewichtung vor einer konkurrierenden Anwendung höherer Gewichtung verschaltet wird.

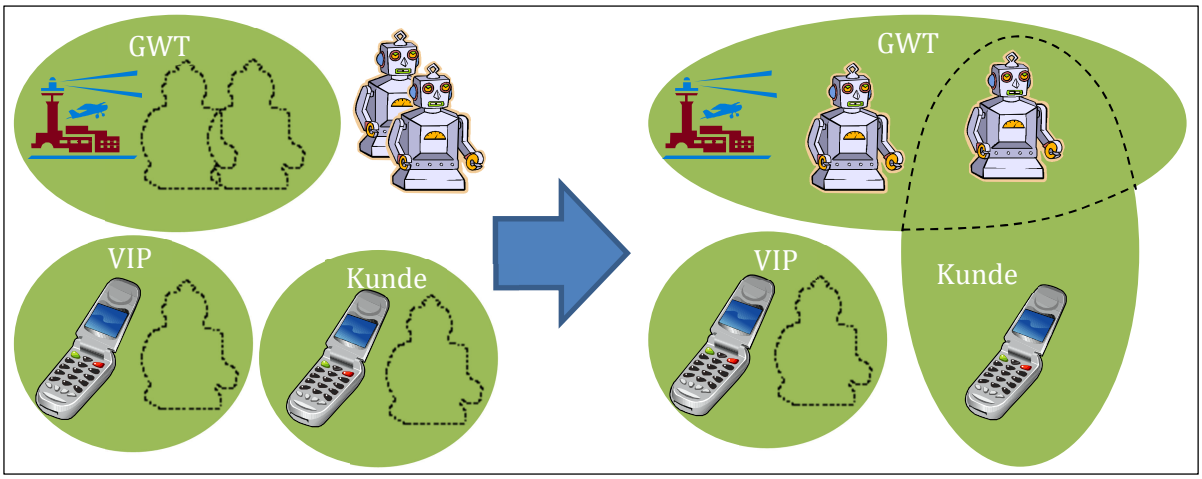

Abbildung 3-11: Szenario 4: Auch unterschiedliche Anwendungen werden gemeinsam verschaltet.

Im 4. Szenario sind die Anwendungen „Kunde“, „Gepäckwagentransport“ (GWT) und „VIP“ aktiv. Zusätzlich existieren zwei freie Roboterkomponenten, zwei Komponenten „Roboter – GUI“ und eine Komponente „Kolonnensteuerung“. Die Anwendungen „Kunde“ und „Gepäckwagentrans-

port“ besitzen beide das gleiche Ziel. Daher sollen diese beiden Anwendungen, wie es in Abbildung 3-11 skizziert ist, mit beiden Robotern verbunden werden.

Mit Szenario 4 besteht die Anforderung, dass nicht nur Anwendungen vom gleichen Typ zusammen eine gemeinsame Anwendungsdomänenkonfiguration bilden können, sondern auch unterschiedliche Anwendungen. Daraus wird die allgemeinere Anforderung abgeleitet, dass eine gemeinsame Anwendungsdomänenkonfiguration aus einer Menge von beliebigen Anwendungen bestehen kann.

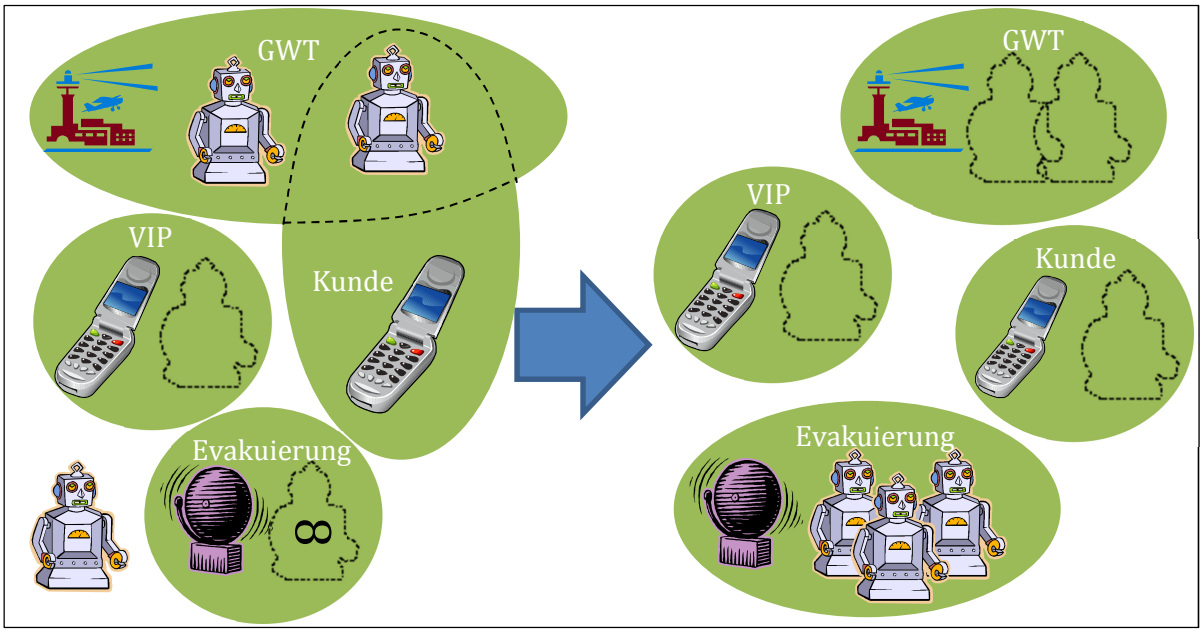

Abbildung 3-12: Szenario 5: Ein Notfallsystem hat vor allen Anwendungen für den Normalbetrieb Vorrang.

Im 5. Szenario wird die Notfallanwendung „Evakuierung“ gestartet. Zu diesem Zeitpunkt existieren in dem System eine freie Roboterkomponente, alle gestarteten Anwendungen aus Szenario 4 mit allen dort beschriebenen Komponenten in der dort abgebildeten Zielsystemkonfiguration. Da ein Notfall immer vor dem Normalbetrieb Vorrang hat, gehört die Notfallanwendung zu einer anderen Klasse von Anwendungen als die übrigen Anwendungen der vorherigen Szenarien. In diesem Szenario müssen sich daher alle Roboter unabhängig davon, ob frei oder gebunden, allein mit dem Notfallsystem verbinden, wie in Abbildung 3-12 dargestellt, um auf den Notfall entsprechend zu reagieren.

Das zuletzt vorgestellte Szenario erweitert die Anforderung von Szenario 2. Es existieren Anwendungen mit einer höheren Priorität für eine Verschaltung im Vergleich zu anderen Anwendungen. Wenn eine solche Anwendung eine Anfrage stellt, kann diese Anwendung auch benötigte gebundene Komponenten von Anwendungskonfigurationen anderer Anwendungen mit geringerer Priorität erhalten. Die nun nicht mehr lauffähigen Anwendungen werden dann zwangsweise pausiert oder abgebrochen. Damit unterscheidet sich die Priorität von der Gewichtung einer Anwendung. Eine Anwendung mit gleicher Priorität und höherer Gewichtung gegenüber einer anderen Anwendung kann von dieser Anwendung keine gebundenen Komponenten erhalten.

Aus den fünf Szenarien wurden fünf funktionale Anforderungen herausgearbeitet. Allerdings muss noch eine nicht funktionale Anforderung aus der Umgebung des Anwendungsbeispiels betrachtet werden. Das Anwendungsbeispiel kommt aus dem Bereich der IT-Ökosysteme. Diese

Systeme bestehen aus einer Vielzahl autonomer Einheiten in einem hoch dynamischen Umfeld. Autonome Einheiten, in diesem Fall Komponenten und Anwendungen, verwalten sich selbst. Daher ist eine zentrale Verwaltung nicht möglich. Unter diesem Aspekt existiert im Flughafen keine zentrale Einheit, welche eine Middleware zur Erstellung einer Systemkonfiguration besitzt. Jede Anwendung ist für die Erstellung der eigenen Anwendungskonfiguration zuständig. Für die Verschaltung ist daher eine dezentrale Middleware notwendig, welche dabei die Konkurrenz in den Szenarien berücksichtigt. Die Middleware muss also in der Lage sein, aus Entscheidungen der einzelnen Anwendungen und ggf. sogar der einzelnen Komponenten ein optimales Ergebnis zu liefern.

Die nachfolgende Liste fasst die einzelnen funktionalen Anforderungen und die nicht funktionale Anforderung kurz zusammen.

FA 1: Jede Anwendung besitzt eine Gewichtung und eine Priorität zu allen anderen Anwendungen.

FA 2: Anwendungen dürfen keine Komponenten von anderen Anwendungen gleicher Priorität erhalten, wenn diese Anwendung ohne die Komponenten nicht mehr laufen kann.

FA 3: Gleiche Anwendungen besitzen als Gruppe eine höhere Gewichtung als einzeln.

FA 4: Unterschiedliche Anwendungen einer Domäne können ebenfalls eine gemeinsame Gruppe bilden.

FA 5: Anwendungen höherer Priorität erhalten bei freien Komponenten den Vorrang und können sogar Komponenten, welche an Anwendungen niedriger Priorität gebunden sind, erhalten.

NFA: Aufbau einer dezentralen dynamisch adaptiven Middleware zur Erstellung der optimalen Systemkonfiguration.

3.4 Anwendung des Komponentenmodells auf das Anwendungsbeispiel

In diesem Abschnitt werden die vorgestellten Komponenten und Anwendungen aus Kapitel 3.2 unter Verwendung des Komponentenmodells aus Kapitel 3.1 formal beschrieben. Anschließend wird die Darstellung der Szenarien an die formale Beschreibung der Komponenten und Anwendungen angepasst.

3.4.1 Beschreibung der Komponenten

Das Komponentenmodell stellt zwei grundlegende Anforderungen an den Aufbau von Komponenten und Anwendungen. Zum einen muss jede Komponente mindestens eine Schnittstelle realisieren. Die Zuordnung einer Komponente zu einer Schablone der Anwendungsbeschreibung wird über die Abgleichung der angebotenen und benötigten Schnittstellen geprüft. Wenn zwei Komponenten unterschiedlicher Semantik die gleichen Schnittstellen benötigen und jeweils keinen Dienst über eine Schnittstelle anbieten, können die Komponenten nicht unterschieden werden. Daher ist es in dem Anwendungsbeispiel notwendig, dass alle Komponenten (Roboter – GUI, Kolonnensteuerung und Fluchtwegsicherung) eine eigene Schnittstelle zur Unterscheidung realisieren. Die zweite Anforderung liegt in der vollständigen Beschreibung der Verschaltungsstruktur einer Anwendung. Das bedeutet, dass für jede Komponente, welche in der Anwendung genutzt werden kann, auch eine passende Schablone zur Zuordnung in der Anwendung vorhanden sein muss.

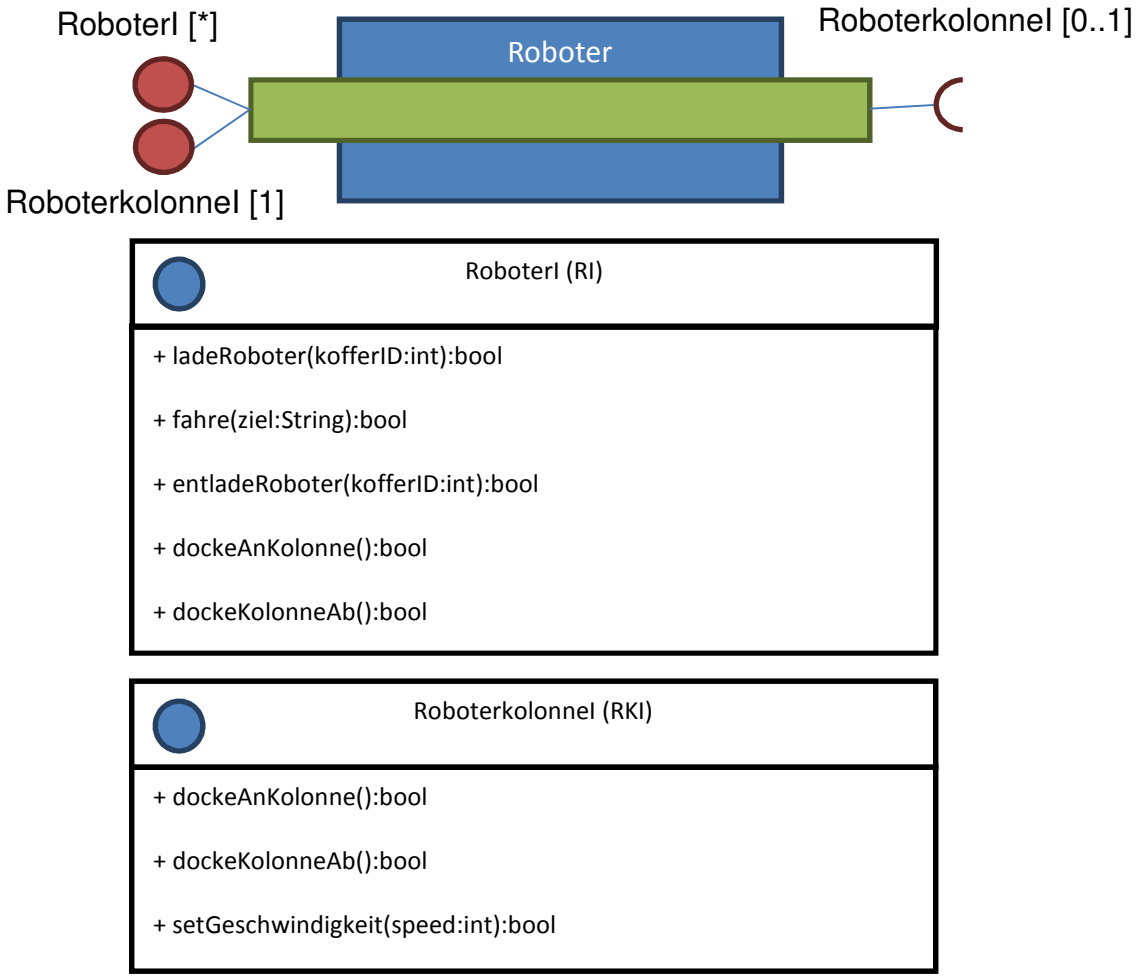

Abbildung 3-13: Darstellung der Roboterkomponente

Die Komponente Roboter aus Abbildung 3-13 realisiert die Kommunikation zu einem physikalischen Transportroboter für Gepäckwagen. Über die Schnittstelle RoboterI bietet die Komponente ihren Dienst anderen Komponenten an. Der Roboter kann Koffer aufnehmen, ein Ziel anfahren und die Koffer dort wieder abstellen. Da die Roboter auch in einer Kolonne arbeiten können, besitzen sie außerdem die Schnittstelle RoboterkolonneI, mit der sich zwei Roboter untereinander koordinieren können. Der vordere Roboter kann dem Roboter hinter ihm eine Geschwindigkeit angeben und durch Lenkmanöver selbst das Ziel ansteuern.

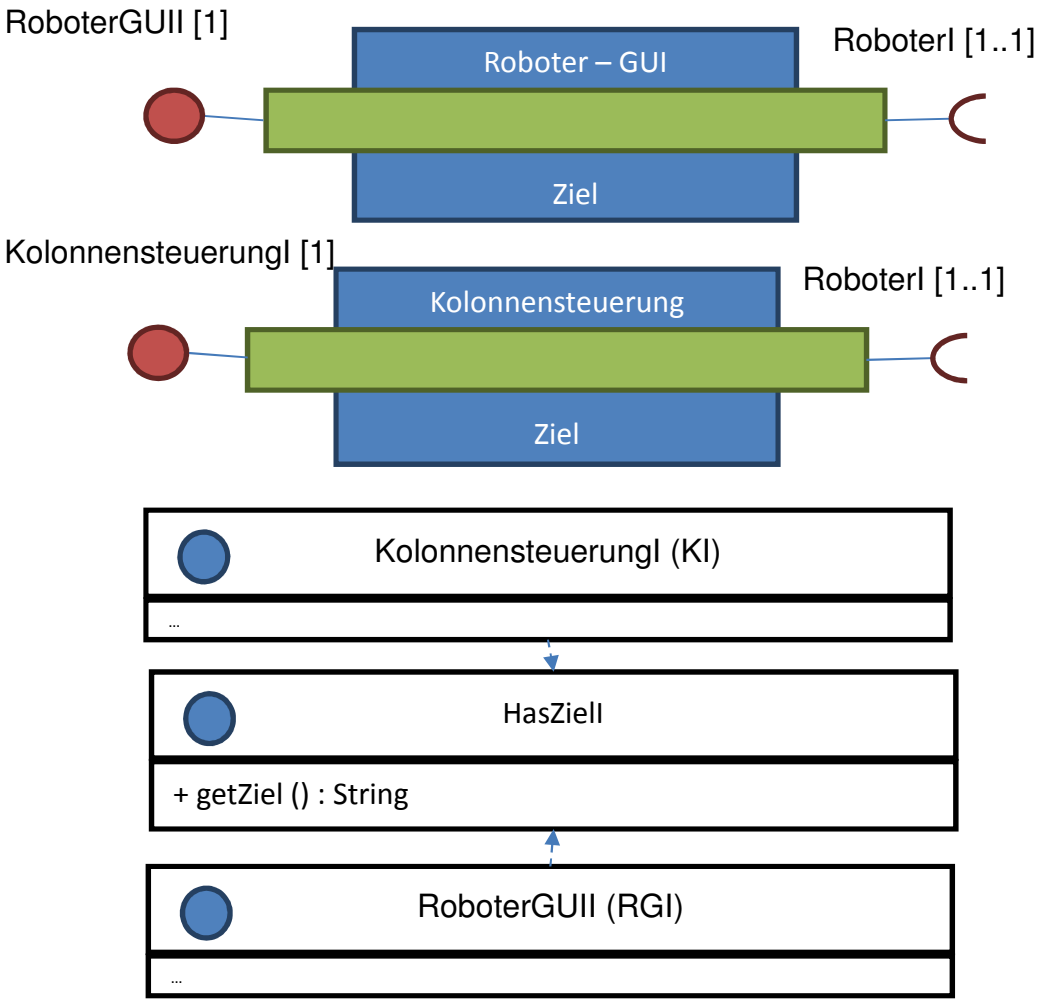

Abbildung 3-14: Darstellung der Komponenten Gepäcktransport und der Kunde

In Abbildung 3-14 werden die Komponenten „Kolonnensteuerung“ und „Roboter – GUI“ mit ihren Schnittstellen dargestellt. Die Komponente „Roboter – GUI“ besitzt eine Benutzerschnittstelle zum Kunden. Die drei Punkte in der zweiten Zeile der tabellarischen Darstellung der Schnittstellen zeigen an, dass diese Schnittstellen Methoden besitzen, welche aber für die Verschaltung nicht benötigt werden und daher ausgeblendet wurden. Diese Schnittstellen sind aber notwendig, um die Komponente der richtigen Anwendung zuordnen zu können. Die Schnittstelle „RoboterGUII“ ist außerdem von der Schnittstelle „HasZielI“ abgeleitet, um während der Verschaltung das Ziel dieser Komponente abrufen zu können. Das Setzen des Ziels zu Beginn ist notwendig, um mehrere Kunden mit gleichem Ziel für denselben Roboter zu verschalten. Die Komponente „Kolonnensteuerung“ besitzt ebenfalls eine Schnittstelle „KolonnensteuerungI“, welche von „HasZielI“ abgeleitet ist. Auch hier soll zu Beginn ein Ziel vorgegeben sein, um eine Verschaltung nach dem Szenario gewährleisten zu können. Dadurch ist es möglich, dass eine Middleware beide Komponenten mit den vorgesehenen Bedingungen für das gleiche Ziel mit demselben Roboter verschalten kann.

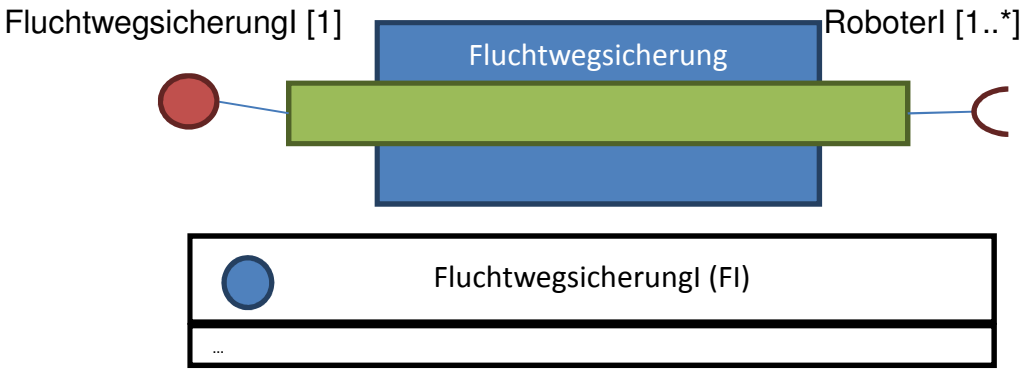

Abbildung 3-15: Darstellung der Notfallkomponente

Die Komponente „Fluchtwegsicherung“ in Abbildung 3-15 nutzt über ihre Komponentenkonfiguration ebenfalls einen Dienst einer Roboterkomponente. Anders als bei den übrigen Komponenten besitzt diese allerdings keine obere Grenze für die Anzahl der zu nutzenden Roboterkomponenten. Der Grund liegt in ihrer Aufgabe, alle Roboter, unabhängig von ihrer Anzahl, dafür einzusetzen, die Fluchtwege für Personen freizuhalten.

3.4.2 Beschreibung der Anwendungen

Nach der Beschreibung der Komponenten können nun die einzelnen Anwendungen der Szenarien, welche die Komponenten benötigen, anhand der Anwendungsbeschreibung aus (Klus 2013) beschrieben werden. Die Anwendungen benötigen ausschließlich die bisher beschriebenen Komponenten.

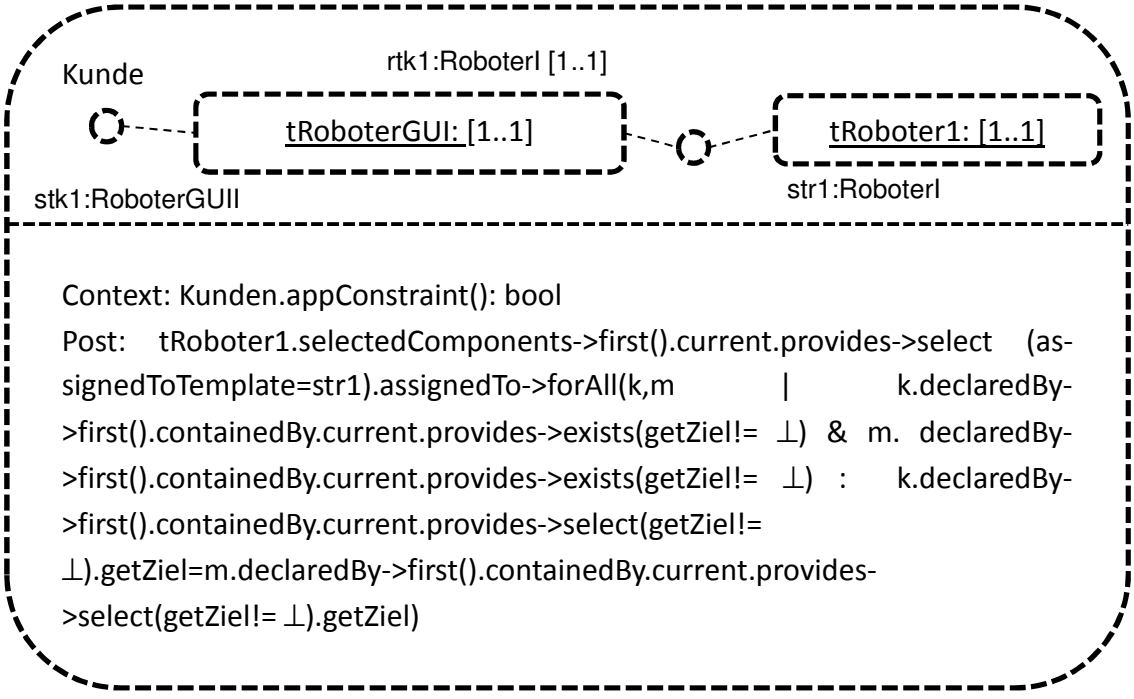

Abbildung 3-16: Anwendungsbeschreibung Kunde

Die Anwendungsbeschreibung „Kunde“ aus Abbildung 3-16 stellt die Verschaltung einer Roboterkomponente mit einer Kundenkomponente dar. Die Anwendungsbeschreibung benötigt genau eine Komponente, welche die Schablone „tRoboterGUI“ erfüllt. Das heißt, die Komponente muss einen Dienst anbieten, welcher die Schnittstelle „RoboterGUII“ implementiert, und einen Dienst über die Schnittstelle „RoboterI“ von genau einer Komponente benötigen. Außerdem

wird eine Komponente für die Schablone „tRoboter1“ benötigt, welche einen Dienst über die Schnittstelle „RoboterI“ anbietet. Über die Bedingung im unteren Bereich der Anwendungsbeschreibung soll sichergestellt werden, dass alle Komponenten, welche diese Roboterkomponente nutzen, das gleiche Ziel besitzen.

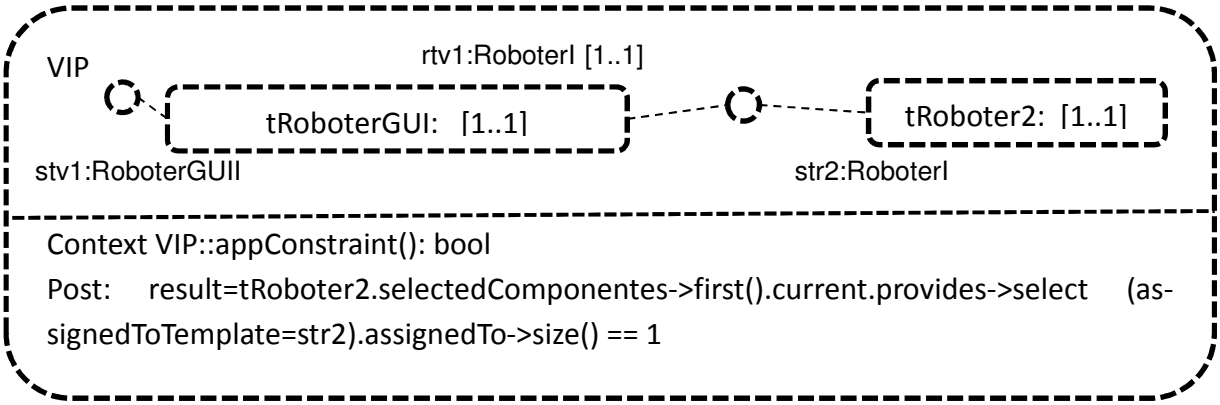

Abbildung 3-17: Anwendungsbeschreibung VIP

Die VIP-Anwendungsbeschreibung in Abbildung 3-17, welche die Nutzung eines Roboters für eine VIP beschreibt, besitzt die gleiche Struktur wie die Kundenanwendungsbeschreibung. Die VIP-Anwendungsbeschreibung benötigt ebenfalls eine Komponente, welche einen Dienst über die Schnittstelle „RoboterGUII“ anbietet und einen Dienst über die Schnittstelle „RoboterI“ benötigt. Außerdem wird eine Komponente benötigt, welche die Schnittstelle „RoboterI“ anbietet. Aber anders als bei der Kundenanwendungsbeschreibung besagt die Bedingung der VIP-Anwendungsbeschreibung, dass der Dienst der Komponente in Template „tRoboter2“ genau von einer Komponente genutzt werden darf. Da dies genau der Verbindung zur Komponente in Template „tRoboterGUI“ entspricht, darf die Komponente, wie im Szenario beschrieben, mit keiner weiteren Komponente über diese angebotene Schnittstelle verschaltet sein.

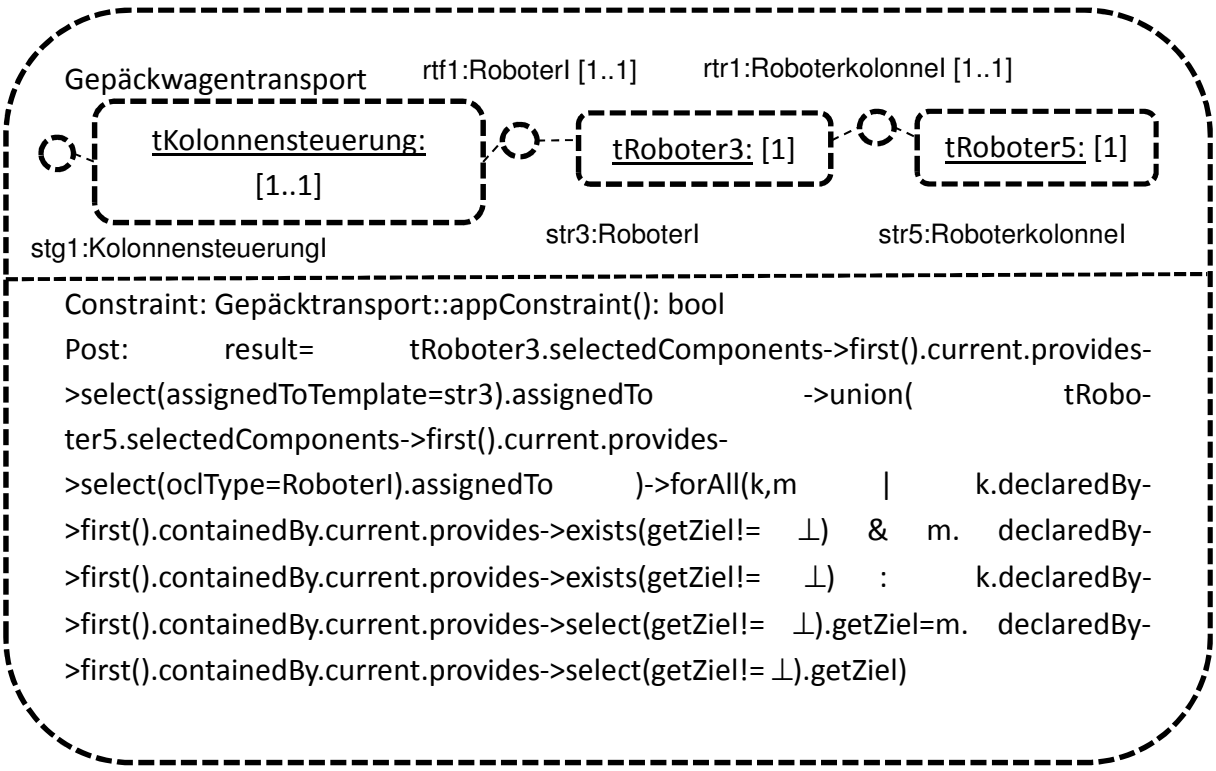

Abbildung 3-18: Anwendungsbeschreibung Gepäckwagentransport

Die Anwendungsbeschreibung in Abbildung 3-18 stellt die Verschaltung für den Gepäckwagentransport dar. Hier werden drei Komponenten benötigt, darunter eine, welche einen Dienst über die Schnittstelle „KolonnensteuerungI“ anbietet und einen Dienst über die Schnittstelle „RoboterI“ benötigt. Außerdem wird eine Komponente benötigt, welche einen Dienst über die RoboterI-Schnittstelle anbietet und einen weiteren Dienst über die Schnittstelle RoboterkolonneI benötigt. Anschließend wird noch eine letzte Komponente benötigt, welche einen Dienst über die Schnittstelle RoboterkolonneI anbietet, aber dafür selbst keinen Dienst benötigt. Die Bedingung der Anwendungsbeschreibung besagt, dass die Komponenten in den Schablonen „tRoboter3“ und „tRoboter5“ über die Schnittstelle RoboterI nur mit Komponenten verbunden sein sollen, welche zusammen alle das gleiche Ziel besitzen.

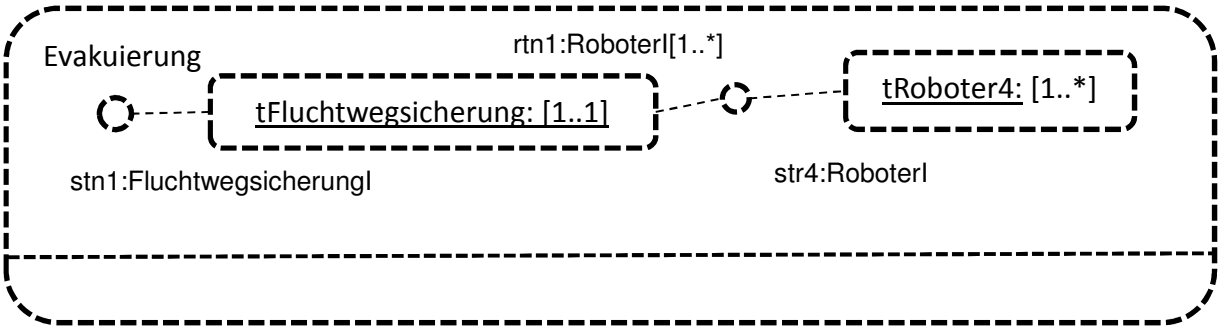

Abbildung 3-19: Anwendungsbeschreibung Evakuierung

Abbildung 3-19 beschreibt die Notfallanwendung „Evakuierung“, die eine Komponente benötigt, welche einen Dienst über die Schnittstelle „FluchtwegsicherungI“ anbietet. Zusätzlich muss die Komponente über einen Dienst eine unbegrenzte Menge an Komponenten nutzen können, welche alle einen Dienst über die Schnittstelle „RoboterI“ anbieten. Außerdem kann eine unbegrenzte Anzahl von Komponenten mit der Schablone „tRoboter4“ verknüpft werden, welche

einen Dienst über die Schnittstelle „RoboterI“ anbieten. Eine Einschränkung durch zusätzliche Bedingungen ist für diese Anwendung nicht vorhanden.

3.4.3 Beschreibung des Anwendungsbeispiels

Die Anwendungen und die dabei verwendeten Komponenten wurden mit dem Komponentenmodell aus Kapitel 3.1 beschrieben. In diesem Abschnitt werden die Szenarien des Anwendungsbeispiels aus Unterkapitel 3.2 mit den Beschreibungen der Komponenten und Anwendungen dargestellt und kurz erläutert.

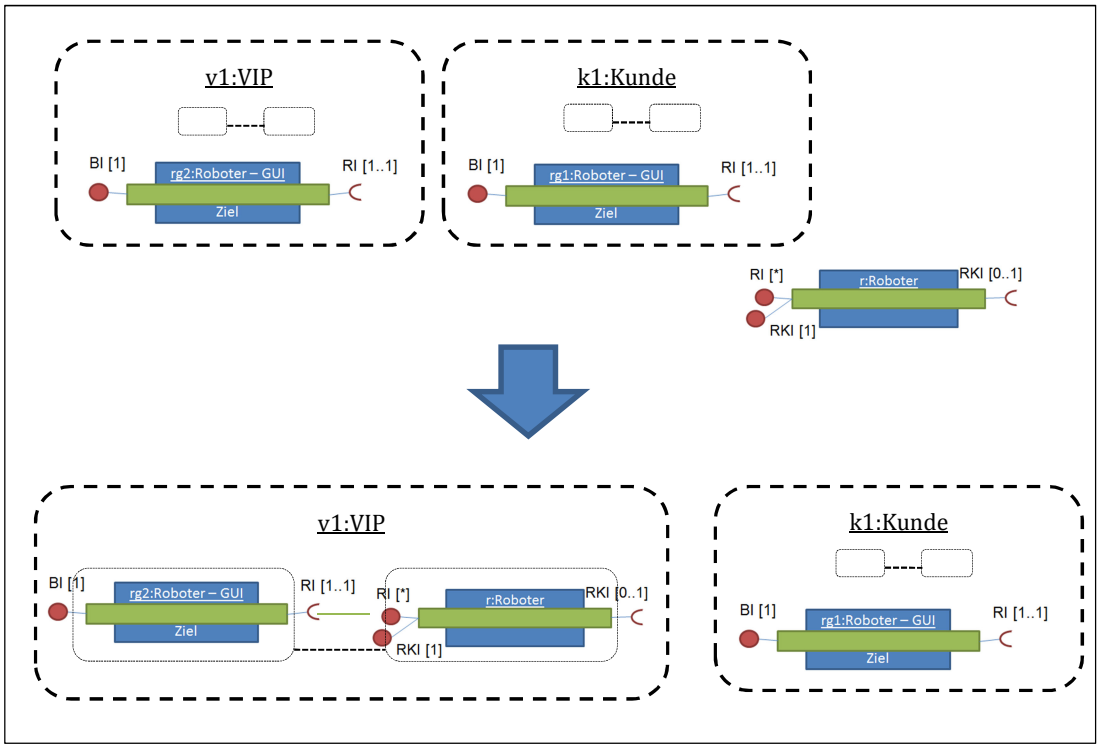

Abbildung 3-20: Szenario 1 unter Verwendung des Komponentenmodells

Analog zur grafischen Darstellung der Szenarien beschreibt auch hier der blaue Pfeil den Übergang einer Systemkonfiguration in eine Zielsystemkonfiguration desselben Systems unter Verwendung einer Middleware. Die Anwendungen werden, analog zu Abbildung 3-3, als gestrichelte Kästchen dargestellt, welches eine Skizze der Struktur, den Namen und den Typ der Anwendung enthält. Die Komponenten werden mittels der Darstellung in Abbildung 3-2 illustriert.

Abbildung 3-20 zeigt das 1. Szenario des Anwendungsbeispiels, in welchem die Anwendung „VIP“ vor der Anwendung „Kunde“ bei Konkurrenz um einen Roboter den Vorrang erhält. Eine Komponente vom Typ „Roboter“ ist ebenso gestartet wie die beiden Anwendungen vom Typ „Kunde“ und „VIP“, welche jeweils eine Komponente vom Typ „Roboter – GUI“ gestartet haben. Wie im Szenario beschrieben, kann die Anwendung vom Typ „Kunde“ mit den Komponenten vom Typ „Roboter – GUI“ und „Roboter“ laufen, wie auch die Anwendung vom Typ „VIP“. Beide Anwendungen können in diesem Szenario nicht dieselbe Roboterkomponente besitzen, da die Anwendungsbeschreibung „VIP“ in ihren Bedingungen dies untersagt. Die Komponente vom Typ „Roboter“, welche mit der Komponente vom Typ „Roboter – GUI“ in der VIP-Anwendung verschaltet ist, darf mit keiner weiteren Komponente verschaltet sein. Wie in Abbildung 3-20 dargestellt, erhält die VIP-Anwendung die Komponente vom Typ „Roboter“.

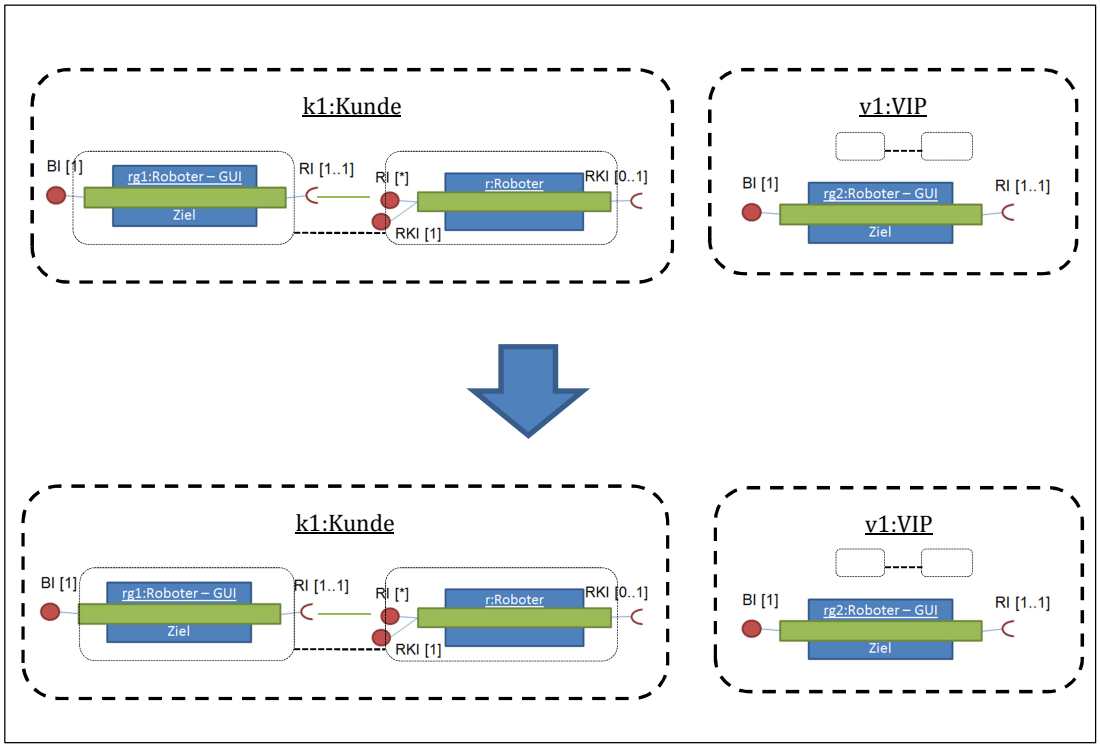

Abbildung 3-21: Szenario 2 unter Verwendung des Komponentenmodells

In Abbildung 3-21 wird dargestellt, dass die Anwendung vom Typ „VIP“ die existierende Roboterkomponente nicht erhält, weil diese Komponente vorher schon mit der Anwendung vom Typ „Kunde“ verbunden wurde. Damit soll sichergestellt werden, dass laufende Anwendungen nicht von anderen Anwendungen durch das Entwenden von Komponenten pausiert oder abgebrochen werden können.

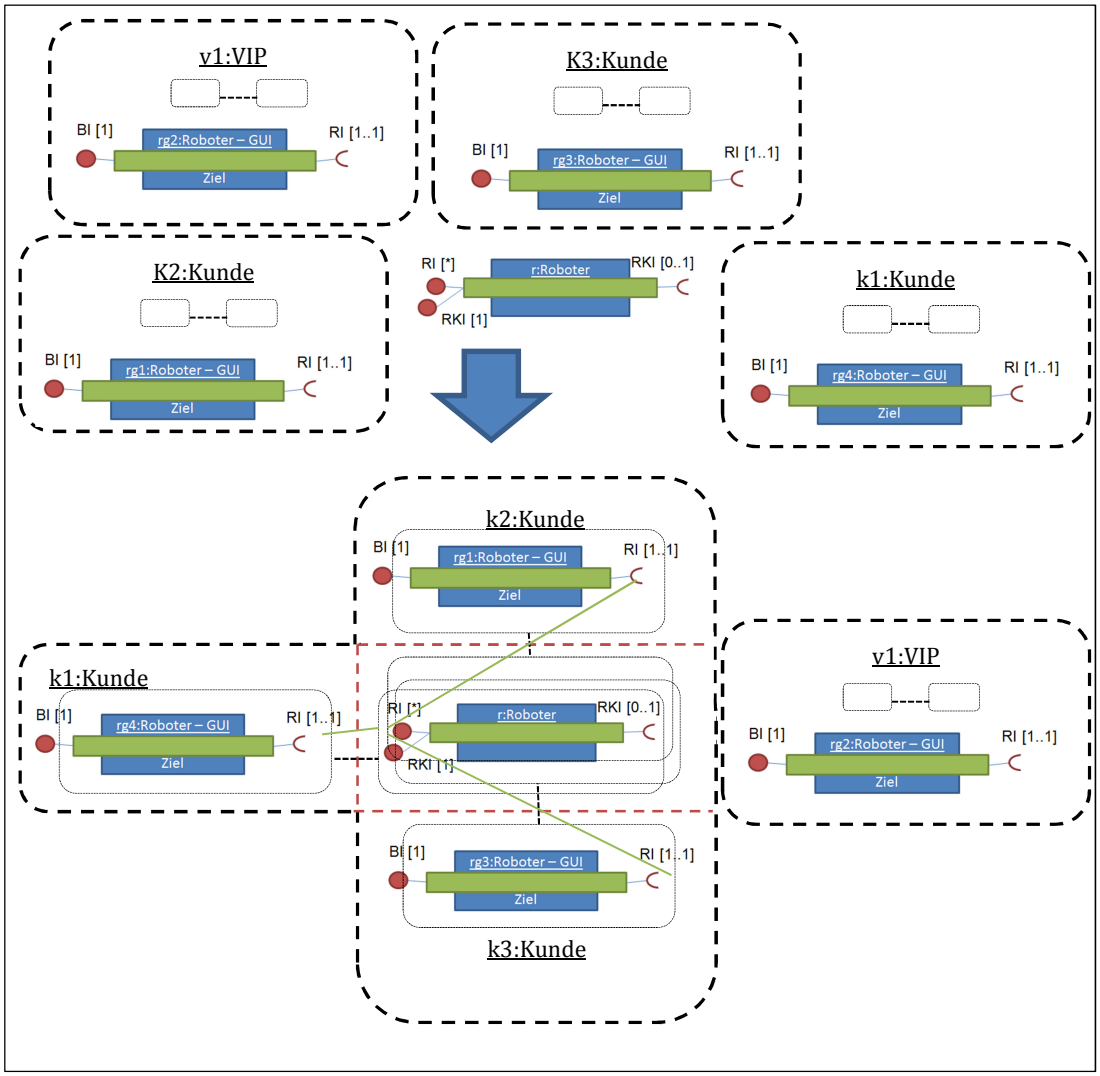

Abbildung 3-22: Szenario 3 unter Verwendung des Komponentenmodells

In Abbildung 3-22 wird das 3. Szenario beschrieben, in welchem drei Kundenanwendungen vor einer Anwendung vom Typ „VIP“ eine Roboterkomponente erhalten. In dieser Abbildung existieren drei Kundenanwendungen und eine VIP-Anwendung mit jeweils einer gestarteten Komponente vom Typ „Roboter – GUI“ für jede der vier Anwendungen und einer freien Roboterkomponente. Wie im Szenario beschrieben, kann die Roboterkomponente entweder mit der VIP-Anwendung oder mit den drei Kundenanwendungen verbunden werden. Prinzipiell gibt es mehrere Möglichkeiten, wie die Roboterkomponente mit den drei Kundenanwendungen verbunden werden kann. Die Roboterkomponente kann mit einer, zwei oder allen drei Anwendungen verbunden werden. Da die Komposition, in der alle drei Kundenanwendungen den Roboter nutzen, aber als beste Variante angesehen wird, besteht nur noch die Entscheidung, die VIP-Anwendung oder die drei Kundenanwendungen zu verschalten. Aufgrund des Szenarios werden die drei Kundenanwendungen vor der VIP-Anwendung verschaltet. Dabei bilden die drei Anwendungskonfigurationen der Kundenanwendungen als Gruppe einer Domäne eine Anwendungsdomänenkonfiguration, wie in Kapitel 3.1.3 erläutert.

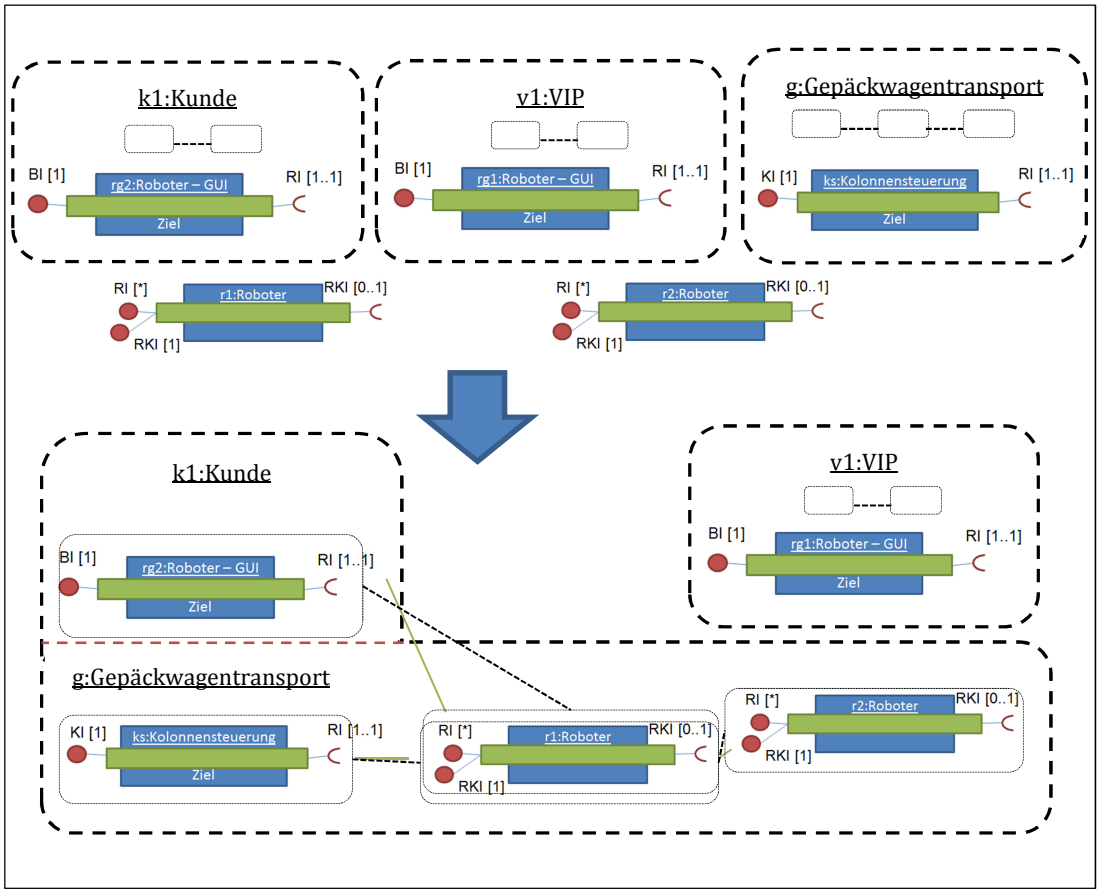

Abbildung 3-23: Szenario 4 unter Verwendung des Komponentenmodells

Das 4. Szenario wird in Abbildung 3-23 dargestellt, in dem eine VIP-Anwendung, eine Gepäckwagentransportanwendung und eine Kundenanwendung um zwei freie Roboterkomponenten konkurrieren. Würden, wie in Abbildung 3-22 dargestellt, nur gleiche Anwendungen zu Anwendungsdomänenkonfigurationen abgebildet, würden die beiden Roboter auf die Anwendungen vom Typ „Kunde“ und „VIP“ aufgeteilt werden. In diesem Fall können zwei Anwendungen laufen, im Gegensatz zu nur der einen Gepäckwagentransportanwendung. Da aber Anwendungen unterschiedlichen Typs dieselbe Komponente verwenden können, soll in diesem Szenario die Gepäcktransportanwendung zusammen mit der Kundenanwendung den Vorrang vor der davor beschriebenen Verschaltungsmöglichkeit erhalten. Die Anwendung vom Typ „Kunde“ nutzt dabei eine der beiden Roboterkomponenten der Gepäckwagentransportanwendung.

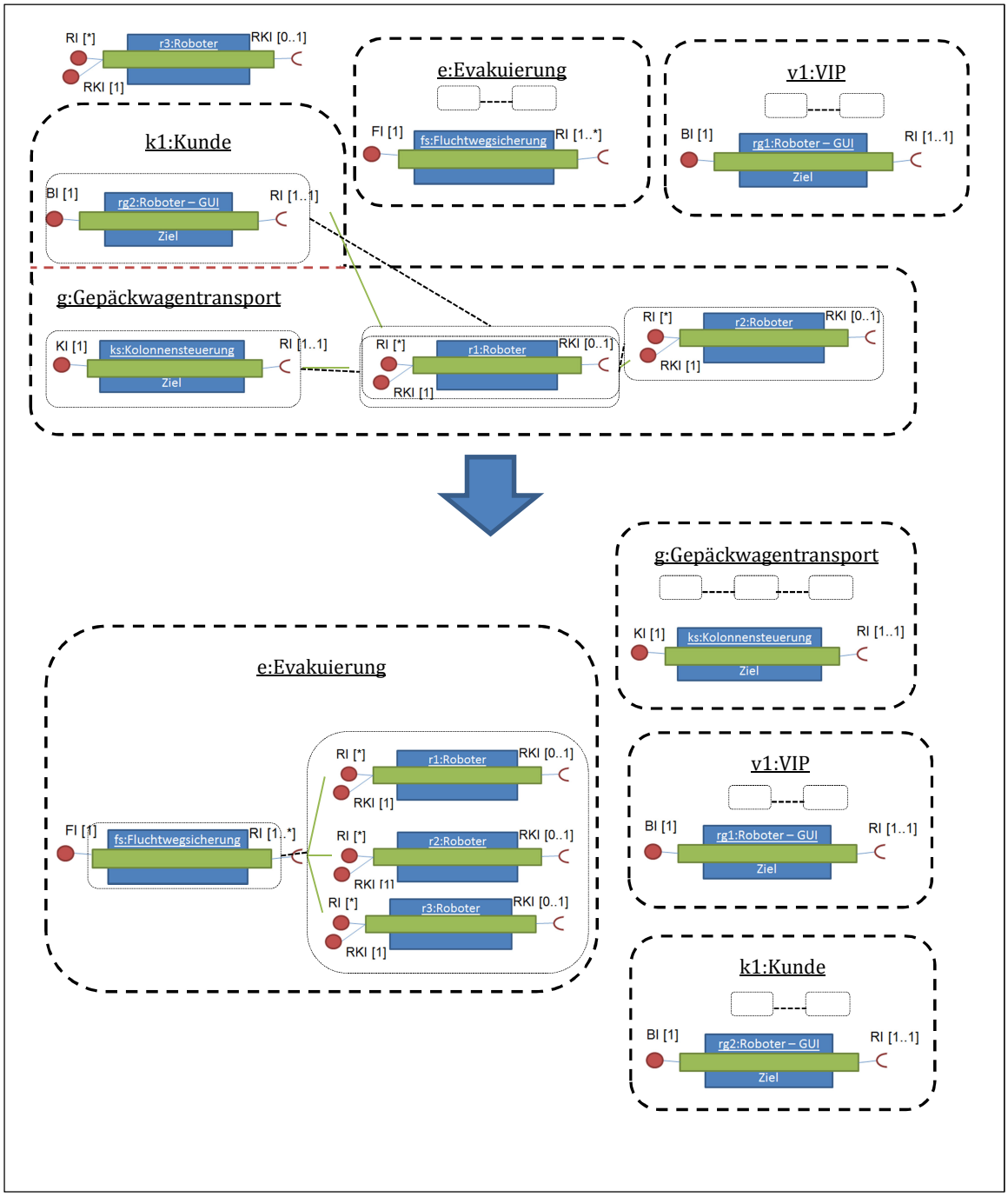

Abbildung 3-24: Szenario 5 unter Verwendung des Komponentenmodells

In Abbildung 3-24 wird das 5. Szenario dargestellt, in dem die Notfallanwendung vom Typ „Evakuierung“ zusammen mit der Komponente vom Typ „Fluchtwegsicherung“ gestartet wird. Zusätzlich existieren noch eine Anwendung vom Typ „VIP“, welche noch eine Roboterkomponente sucht, und die Systemkonfiguration mit den Anwendungen vom Typ „Gepäcktransport“ und „Kunde“ aus Abbildung 3-23 mit ihren Komponenten. Außerdem befindet sich im System noch eine freie Roboterkomponente. Anders als in Abbildung 3-21 reagieren alle Roboterkomponenten auf die Anfrage der Notfallanwendung. Alle existierenden Roboterkomponenten werden anschließend der Anwendung vom Typ „Evakuieren“ zugeordnet. Die Anwendungen vom Typ „Gepäckwagentransport“ und „Kunde“ müssen demnach ihre Arbeiten pausieren bzw. abbrechen.

3.5 Problemstellung und Forschungsfragen

Mit dem präsentierten Komponentenmodell aus Kapitel 3.1 können die Komponenten und Anwendungen des Anwendungsbeispiels vollständig beschrieben werden. Auch die Szenarien können zumindest dargestellt werden.

Allerdings fehlt in dem in (Klus 2013) beschriebene Ansatz die Möglichkeit, eine Systemkonfiguration zu bewerten, um die erwartete optimale Systemkonfiguration der jeweiligen beschriebenen Szenarien bestimmen zu können. Die funktionale Anforderung aus Szenario 1 besagt, dass jede Anwendung eine Gewichtung besitzen muss. Damit kann z. B. entschieden werden, dass die VIP-Anwendung im direkten Vergleich zu einer einzelnen Kundenanwendung bei Konkurrenz um eine Roboterkomponente Vorrang bekommen soll. In dem Ansatz in (Klus 2013) existiert aber kein Konzept zur Beschreibung einer Gewichtung.

Die funktionale Anforderung 2 wird ebenfalls nicht unterstützt. Sie besagt, dass verbundene Komponenten von anderen Anwendungen gleicher Priorität keine Komponenten erhalten dürfen, wenn die anderen Anwendungen dann nicht mehr lauffähig sind. Da in (Klus 2013) nur eine Anwendung betrachtet, tritt dieser Fall bei ihm nicht auf, weshalb dazu auch keine Aussage getroffen wurde.

Die 3. funktionale Anforderung fordert eine Gruppierung von mehreren gleichen Anwendungen einer Anwendungsdomäne zur Kombinierung ihrer Gewichtungen zu einer höheren Gewichtung. Analog zur funktionalen Anforderung 1 können hier ohne ein Konzept für die Gewichtung auch keine Gewichtungen kombiniert werden. Die funktionale Anforderung 4, Gruppierung von unterschiedlichen Anwendungen einer Anwendungsdomäne zu einer Anwendungsdomänenkonfiguration, ist aus diesen Gründen ebenfalls nicht möglich.

Bei der letzten funktionalen Anforderung sollen Anwendungen höherer Priorität bei allen freien Komponenten vor Anwendungen geringerer Priorität Vorrang bekommen. Auch sollen die Anwendungen höherer Priorität Komponenten von Anwendungen geringerer Priorität erhalten können, unabhängig davon, ob die Anwendungen geringerer Priorität ohne diese Komponenten weiter laufen können. Bei dem Ansatz in (Klus 2013) existiert, wie für die Gewichtung, auch kein Konzept zur Beschreibung einer Priorisierung einer Anwendung, weshalb Anwendungen in seinem Ansatz nicht unterschieden werden können.

Die nicht funktionale Anforderung fordert für die Verschaltung einen dezentralen Algorithmus. Der in (Klus 2013) beschriebene Ansatz verwendet zwar einen dezentralen Algorithmus zur Verschaltung einer Anwendung. Allerdings muss dieser Algorithmus nun auf die Verschaltung mehrerer untereinander konkurrierender Anwendungen angepasst werden.

Wenn die Anwendungen die Informationen, welche in den funktionalen Anforderungen benötigt werden, bereitstellen, kann anschließend eine Funktion entwickelt werden, um eine Systemkonfiguration oder eine Anwendungsdomänenkonfiguration zu bewerten. Diese Bewertungsfunktion kann dann für die Erforschung eines Algorithmus verwendet werden, welcher die Komponenten dezentral zu der optimalen Systemkonfiguration verschalten kann. Die Arbeit beschäftigt sich mit diesem Problem, welches im nächsten Satz in Form einer Frage zusammengefasst ist:

Wie kann eine Menge von konkurrierenden Anwendungen dezentral zu einer optimalen Systemkonfiguration verschaltet werden?

Zur Bearbeitung des Problems ist eine Reihe von Forschungsfragen zu beantworten:

Wie kann die optimale Systemkonfiguration eines Systems definiert werden?

Wie ist ein dezentraler Algorithmus aufgebaut, der diese optimale Systemkonfiguration immer erreicht?

Wie kann die Stabilität des Algorithmus und des erreichten Optimums sichergestellt werden?

Wie kann die entwickelte dezentrale Middleware evaluiert werden?

Im nächsten Unterkapitel werden verwandte Arbeiten beschrieben, welche sich mit Teilen des Problems beschäftigt haben, aber das Problem als Ganzes nicht ausreichend lösen können. Anschließend wird in den darauffolgenden Kapiteln die für dieses Problem entwickelte Lösung beschrieben.

3.6 Verwandte Arbeiten

Betrachtet werden zwei Bereiche, welche sich mit der dezentralen Verteilung von Ressourcen beschäftigen. Der eine Bereich besteht aus dynamisch adaptiven komponentenbasierten Softwaresystemen und der andere Bereich aus Multiagentensystemen.

3.6.1 Dynamisch adaptive komponentenbasierte Softwaresysteme

Das Framework PCOM (Becker et al. 2004) ermöglicht die Entwicklung von selbstadaptiven verteilten Systemen und Anwendungen. Wie schon in den Grundlagen in Kapitel 2 beschrieben, werden diese Systeme und Anwendungen aus Komponenten aufgebaut. Die Kommunikation zwischen zwei Komponenten wird über Verträge spezifiziert. Über diese kann PCOM passende Komponenten zur Laufzeit zu einer Anwendung verschalten.

Der Aufbau einer Anwendung entspricht dem einer Baumstruktur. Die Wurzel wird von einer Anwendungskomponente repräsentiert. Sie enthält anwendungsspezifische Funktionen und realisiert z. B. die Schnittstelle zum Benutzer. Ausgehend von der Anwendungskomponente werden nun von PCOM rekursiv für jede Komponente die benötigten Komponenten gesucht, welche den Schnittstellenbeschreibungen der Verträge der jeweiligen Komponente entsprechen.

Auch auf Änderungen der Umgebung oder des Systems, z. B. durch das Hinzufügen oder Wegfallen von Komponenten, kann PCOM reagieren. Es konfiguriert die Anwendungen dynamisch neu, damit diese weiterhin genutzt werden können.

Allerdings betrachtet PCOM keine Konflikte, die auftreten, wenn zwei Anwendungen dieselbe Komponente benötigen. Daher kann die Problemstellung dieser Arbeit in diesem Framework nicht gelöst werden.

Die Middleware COMITY (Tuttlies et al. 2007) ist eine Erweiterung von PCOM. Sie beschreibt ein System, bei welchem Anwendungen bei der Wahl ihrer benötigten Komponenten auf Bedingungen anderer Anwendungen achten, um Konflikte zu vermeiden.

Wie schon bei PCOM beschrieben, kann jede Komponente exklusiv nur von einer Komponente und von einer Anwendung verwendet werden. Daher wird der Anwendungsfall, dass eine Komponente von mehreren Anwendungen genutzt wird, nicht betrachtet. Die Konfliktbehandlung

betrifft nur Kontextinformationen der Anwendungen und der aktuellen Umgebung. Somit kann auch diese Middleware das Problem, welches von dem Anwendungsbeispiel dieser Arbeit illustriert wird, nicht lösen.

Mit der Middleware „Accord“ (Liu und Parashar 2006) aus dem Projekt AutoMate (Parashar et al. 2006) können Komponenten zu Anwendungen unter Berücksichtigung einer Anwendungsbeschreibung dezentral verschaltet werden. Für die Auswahl der geeigneten Komponenten werden vom Entwickler Regeln definiert. Die Verschaltung der Komponenten wurde über eine agentenbasierte Infrastruktur realisiert.

In der Middleware werden allerdings keine Verschaltungskonflikte zwischen den Anwendungen beschrieben. Eine Regel zur Gewichtung oder Priorisierung von Anwendungen untereinander ist auch nicht enthalten. Zudem fehlt eine Definition der optimalen Systemkonfiguration, weshalb auch diese Middleware das in diesem Kapitel beschriebene Problem nicht lösen kann.

Die Middleware „FlashMob“ (Sykes et al. 2011) ermöglicht ebenfalls eine dezentrale Konfiguration von Komponenten zu einer Anwendung. Diese Middleware ist in drei Schichten unterteilt. Die unterste Schicht enthält eine feste Menge von gestarteten Komponenten und kann eine existierende Konfiguration aus der mittleren Schicht umsetzen. Die mittlere Schicht erstellt dezentral eine Konfiguration aus der Anzahl der Komponenten und den Anforderungen dieser Anwendung aus der oberen Schicht. Die obere Schicht enthält alle funktionalen und nicht funktionalen Anforderungen dieser Anwendung. Sowohl die mittlere als auch die untere Schicht arbeiten dabei verteilt.

Diese Middleware ist in der Lage, eine Anwendung dezentral zu verschalten. Allerdings wird keine Aussage zur Konkurrenz zu anderen Anwendungen getroffen. In dem Anwendungsbeispiel dieser Arbeit muss bei der Wahl einer Komponente geprüft werden, ob die Anwendungen, welche diese Komponente ebenfalls nutzen möchten, irgendwelche Bedingungen verletzen. Als Beispiel kann die Kundenanwendung genommen werden. Sie setzt ein Ziel für den Roboter fest und die Bedingung, dass alle Anwendungen, welche denselben Roboter verwenden, ebenfalls das gleiche Ziel besitzen müssen. Diese Bedingung tritt während der Verschaltung auf und muss daher bei der Erstellung einer Konfiguration geprüft werden, was die Middleware „FlashMob“ nicht berücksichtigt. Daher kann das Problem dieser Arbeit mit dieser Middleware nicht abgebildet werden.

3.6.2 Multiagentensysteme

Aus dem Bereich der Multiagentensysteme existieren ebenfalls Arbeiten, welche sich mit der dezentralen Zuordnung von Ressourcen beschäftigen. Mit dem System „Challenger“ (Chavez et al. 1997) können Prozesse, welche auf einem Rechner ausgeführt werden, auf den Rechnerressourcen eines anderen Rechners im Netzwerk berechnet werden. Auf jedem dieser Rechner existiert ein Agent, welcher über ein Auktionsverfahren denjenigen Rechner im Netzwerk identifiziert, der den Prozess am schnellsten abwickeln kann. Dieser Rechner kann auch derjenige sein, auf dem dieser Agent läuft. Jeder Prozess kann dabei nur auf einem Rechner ausgeführt werden, eine Aufteilung auf mehrere Rechner ist im Konzept des Systems nicht vorgesehen. Zusätzlich besitzen die identischen Agenten auch noch lernende Eigenschaften. Ein Agent besitzt z. B. im Vergleich zu anderen Agenten die kürzeste Berechnungszeit für den angefragten Prozess. Aber in der Vergangenheit hat dieser häufig relativ lange zum Antworten auf Anfragen gebraucht. Dies wird in die Entscheidung mit eingebracht und der Agent entscheidet sich dann ggf. für einen anderen Agenten.

Dieser Ansatz bildet einen Prozess auf genau eine Ressource ab. Bei dem Problem dieser Arbeit können Komponenten als begrenzte Ressource hingegen mehreren Anwendungen zugewiesen werden und eine Anwendung benötigt meist mehr als eine Komponente, um zu laufen. Daher kann weder das Problem noch das Anwendungsbeispiel mit diesem Ansatz abgebildet werden. Des Weiteren stellen Anwendungen in Bezug auf die geeignete Auswahl an Komponenten auch Bedingungen an diese Komponenten, weshalb der benötigte Abstimmungsprozess mit dem System „Challenger“ nicht realisiert werden kann.

In der Arbeit von Netzer (Netzer et al. 2015) werden Agenten beschrieben, welche zur Laufzeit eine Menge von Ressourcen untereinander verteilen. Das Optimum der Verteilung soll dem globalen Minimum des Neids untereinander entsprechen. Ein Agent besitzt gegenüber einem anderen Agenten einen Wert für Neid für jede Ressource, welche der andere Agent besitzt und der erste Agent benötigt. Ressourcen werden in diesem Ansatz als unteilbare Einheit definiert, welche immer genau einem Agenten zugeordnet sein soll. Für eine geringe Menge an Agenten und Ressourcen wird das Optimum durch das dezentrale Herstellen aller möglichen Verteilungen der Ressourcen zu den Agenten gefunden. Ab einer einstellbaren Menge von Agenten und Ressourcen ist dieses Verfahren nicht mehr in akzeptabler Zeit umsetzbar. Deshalb wird auf eine ausreichend gute Lösung über lokale Optimierung von Teilmengen von Agenten gewechselt.

Dieser Ansatz kann auf den Bereich der Komponentenverschaltung abgebildet werden. Somit entspricht ein Agent einer Anwendung und eine Ressource entspricht einer Komponente, welche einer Anwendung zugeordnet werden kann. Aber anders als im Ansatz kann eine Komponente von mehreren Anwendungen genutzt oder auch keiner Anwendung zugeordnet werden. Wie in dem Anwendungsbeispiel in diesem Kapitel dargestellt, existieren auch zwischen den Anwendungen Bedingungen. So besitzen z. B. alle Anwendungen, welche in dem Anwendungsbeispiel denselben Gepäckwagenroboter nutzen, das gleiche Ziel. Dieser Aspekt kann in dem in (Netzer et al. 2015) beschriebenen Ansatz nicht abgebildet werden. Auch ändert sich die Menge an Neid gegenüber einem anderen Agenten schon bei Transformation einer einzelnen Ressource zwischen zwei Agenten. Eine Anwendung kann allerdings erst laufen, wenn alle nötigen Komponenten ihr zugeordnet sind. Ob eine oder alle nötigen Komponenten einer Anwendung fehlen, macht für sie keinen Unterschied, weshalb viele Einzeltransformationen hier keinen sichtbaren Effekt besitzen, auf dem das Verfahren des Ansatzes aufbauen kann. Dieser Ansatz eignet sich daher ebenfalls nicht zur Darstellung des Problems dieser Arbeit.

4 Überblick der Lösung

Im vorherigen Kapitel wurde anhand eines Anwendungsbeispiels die Problemstellung herausgearbeitet. In diesem Kapitel wird für die Problemstellung ein Lösungsansatz erstellt. Dazu wird jede Forschungsfrage der Problemstellung aus Kapitel 3 in diesem Kapitel einzeln analysiert und eine Antwort auf diese Frage erarbeitet.

Das Anwendungsbeispiel aus dem 3. Kapitel beschreibt Anwendungen, die untereinander um benötigte Komponenten konkurrieren. Allerdings anders als bei (Klus 2013) beinhaltet das System, in dem das Anwendungsbeispiel aus Kapitel 3 dargestellt werden soll, mehrere Anwendungen und mehrere Instanzen einer Anwendung. In dem bestehenden Modell aus Abbildung 3-1 werden Komponenten einer Anwendung zugeordnet. Die Existenz mehrerer Instanzen einer Anwendung, wo jede Instanz eine eigene Menge von Komponenten erhält, ist in dem Modell nicht beschrieben. Folglich können die Szenarien des Anwendungsbeispiels nicht korrekt mit dem Modell beschrieben werden. Da in manchen Szenarien allerdings mehrere Instanzen derselben Anwendung auftauchen, muss dies auch in dem zugehörigen Metamodell beschrieben werden können. Daher gilt es das Komponentenmodell dementsprechend zu erweitern.

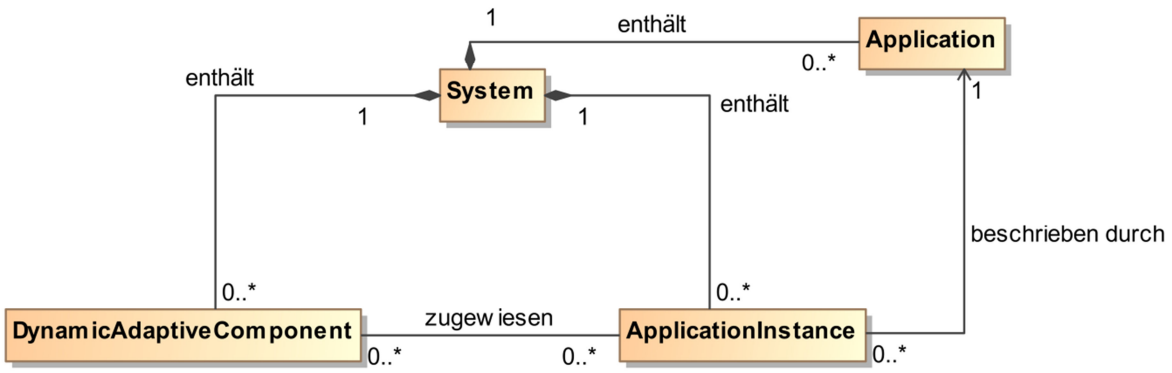

Abbildung 4-1: Grobe Sicht auf das Metamodell

In Abbildung 4-1 wird ein Metamodell dargestellt, welches auf die wichtigsten Klassen beschränkt ist. Darin werden nur die vier Klassen „System“, „Application“, „ApplicationInstance“ und „DynamicAdaptiveComponent“ dargestellt. „Application“ und „DynamicAdaptiveComponent“ entsprechen den gleichnamigen Klassen aus dem Komponentenmodell aus (Klus 2013). „ApplicationInstance“ stellt eine Instanz einer Anwendung (Application) dar und die Klasse „System“ steht für das bestehende System, zu dem anschließend die Systemkonfiguration beschrieben wird. Das System besteht aus allen Anwendungen, allen Anwendungsinstanzen (ApplicationInstance) und allen Komponenten (DynamicAdaptiveComponent). Anwendungen, Anwendungsinstanzen und Komponenten sind dabei genau einem System zugeordnet und jede Anwendungsinstanz wird durch genau eine Anwendung beschrieben. Mithilfe dieses Metamodells kann nun eine optimale Systemkonfiguration definiert werden.

4.1 Definition einer optimalen Systemkonfiguration

In der ersten Forschungsfrage wird eine Definition einer optimalen Systemkonfiguration eines Systems gesucht. Aufgrund der Anforderungen aus dem Anwendungsbeispiel werden für eine optimale Systemkonfiguration die Gewichtung der Anwendungskonfigurationen und die Priorität der Anwendungen benötigt. Beide Eigenschaften sind in dem in (Klus 2013) beschriebenen

Modell ebenfalls nicht vorhanden, siehe Abbildung 3-1. Daher muss das Modell auch dahin gehend erweitert werden. Diese Definition ermöglicht anschließend eine Sortierung aller möglichen Systemkonfigurationen unter Verwendung der Gewichtungen der Anwendungskonfigurationen der Anwendungen im System. Außerdem wird die Anforderung, dass höher priorisierte Anwendungen immer vor geringer priorisierten Anwendungen Vorrang haben, in der Definition der Systemkonfiguration berücksichtigt. Systemkonfigurationen, welche diese Anforderung verletzen, werden damit als ungültig definiert.

4.1.1 Definition des Systems

Bevor allerdings die optimale Systemkonfiguration eines Systems formal beschrieben werden kann, muss das dabei verwendete System definiert sein. Das Metamodell dieses Systems basiert auf dem Komponentenmodell von Holger Klus, welches schon in Kapitel 3 beschrieben wurde. Da dieses Modell aber weder mehrere Instanzen einer Anwendung noch die Gewichtung oder Priorisierung von Anwendungen beschreibt, wird dieses Modell wie in Abbildung 4-2 erweitert. Auf Basis des Metamodells in Abbildung 4-1 wurde nun ein verfeinertes Metamodell mit allen Klassen aus dem Komponentenmodell aus Kapitel 3 erstellt.

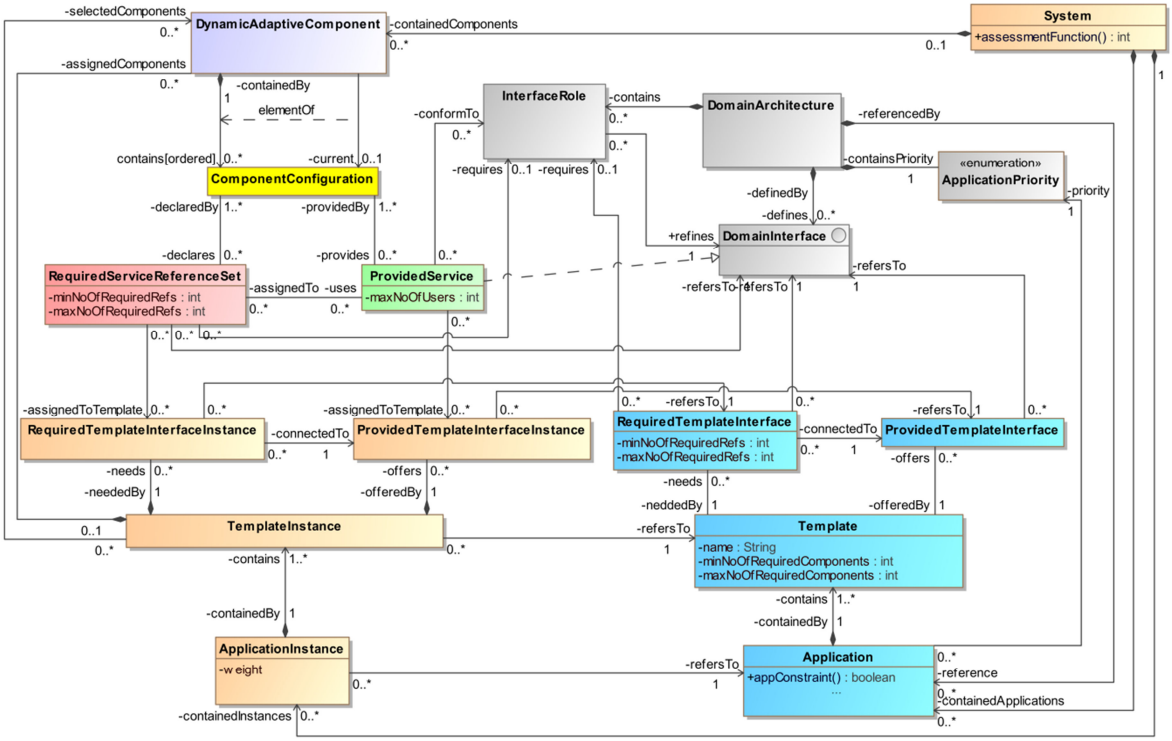

Abbildung 4-2: Verfeinertes Metamodell

Alle Klassen und die meisten Assoziationen wurden von dem Komponentenmodell aus Kapitel 3 in das Metamodell in Abbildung 4-2 übernommen. Zum einen wurden die komplette Struktur einer Komponente bestehend aus den 4 Klassen „DynamicAdaptiveComponent“, „ComponentConfiguration“, RequiredServiceReferenceSet“ und „ProvidedService“ und deren Beziehungen zueinander übertragen. Zum anderen wurde die komplette Struktur einer Anwendung bestehend aus ebenfalls 4 Klassen, „Application“, „Template“, „RequiredTemplateInterface“ und „ProvidedTemplateInterface“, mit deren Beziehungen zueinander übertragen. Auch die Domänenschnittstellen (DomainInterface), welche wie die Anwendungen einer Anwendungsdomäne (DomainArchitecture) zugeordnet sind, wurden übertragen, ebenso die Schnittstellenrolle (InterfaceRole). Die Domänenschnittstellen stellen sicher, dass ausschließlich Komponenten den

jeweiligen Schablonen zugeordnet werden, welche auf die zu einem Template passenden Domänenschnittstellen verweisen.

Entfernt wurden bei der Übertragung die direkten Beziehungen zwischen der Anwendungsstruktur und der Komponentenstruktur, weil die Beziehungen nun über die Struktur der Anwendungsinstanz aufgebaut werden, siehe Abbildung 4-1. Die Struktur einer Anwendungsinstanz ist analog zur Struktur einer Anwendung aufgebaut. Damit kann jede Instanz mit einer eigenen Menge an Komponenten befüllt werden. Sie besteht aus den 4 Klassen „ApplicationInstance“, „TemplateInstance“, „RequiredTemplateInterfaceInstance“ und „ProvidedTemplateInterfaceInstance“, welche jeweils auf ihre Entsprechung in der Anwendungsstruktur verweisen (z. B. TemplateInstance->Template). Außerdem existieren nun zwei Assoziationen von „TemplateInstance“ zu „DynamicAdaptiveComponent“. Die Assoziation „selectedComponents“ beschreibt, wie auch die gleichnamige Assoziation bei dem Modell in (Klus 2013), die Komponenten, welche zur Laufzeit dieser Anwendungsinstanz zugeordnet sind. Mit der Assoziation „assignedComponents“ können die Komponenten beschrieben werden, welche exklusiv immer dieser Anwendungsinstanz zugeordnet sind. In dem Anwendungsbeispiel wird z. B. die Komponente „Roboter-GUI“ einer Anwendungsinstanz vom Typ „Kunde“ darüber zugeordnet. Da diese Komponenten von dem Lösungsverfahren nicht einer Anwendungsinstanz dynamisch zugeordnet werden sollen, müssen diese in dem Modell unterschieden werden.

Daher entspricht die in Kapitel 3 beschriebene Anwendungskonfiguration einer Anwendungsinstanz nun der Vereinigung der Mengen an Komponenten dieser beiden Assoziationen. Weiterhin gilt, dass eine laufende Anwendungsinstanz eine Anwendungskonfiguration besitzt, bei der die Methode „appConstraint“ der über „refersTo“ zugeordneten Anwendung (Application) erfüllt ist, was in Formel 4-1 dargestellt ist.

$$ApplicationIsRunning: ApplicationInstance \rightarrow Bool$$

$$Application\text{IsRunning}(\text{appInstance}) \Leftrightarrow \text{appInstance.contains.selectedComponents} \neq \emptyset \wedge \text{appInstance.refersTo.appConstraint}() \quad (4\text{-}1)$$

Auch wurde die Klasse „System“ aus Abbildung 4-1 mit ihren Assoziationen abgebildet. Die zugehörige Systemkonfiguration aus Kapitel 3 ist nun ebenfalls im Metamodell enthalten. Sie besteht in der vereinigten Menge aller Komponenten der in den Anwendungskonfigurationen der vom System über „containedInstances“ verbundenen Anwendungsinstanzen. Jede im System existierende Anwendungskonfiguration ist demnach eine Teilmenge der Systemkonfiguration. Außerdem wurden die Enumeration „ApplicationPriority“, eine Beziehung zwischen der Klasse „Application“ und der Enumeration „ApplicationPriority“, und das Attribut „weight“ der Klasse „ApplicationInstance“ hinzugefügt.

Das Attribut „weight“ liefert die Gewichtung einer Anwendungsinstanz. Mit ihr lassen sich Anwendungsinstanzen untereinander sortieren. Als Beispiel können die Instanzen der Anwendungen „Kunde“ und „VIP“ des Anwendungsbeispiels betrachtet werden. Die VIP-Anwendungsinstanz besitzt einen höheren Wert als die Kundenanwendungsinstanz, damit dadurch die Zielsystemkonfiguration von Szenario 1 des Anwendungsbeispiels abgebildet werden kann. Da unterschiedliche Anwendungskonfigurationen einer Anwendungsinstanz unterschiedliche Gewichtungen haben können, ist die Gewichtung außerdem von der zugehörigen Anwendungskonfiguration abhängig. Über die Beziehung „priority“ sind die Anwendungen ebenfalls jeweils einer Priorität (ApplicationPriority) fest zugeordnet.

Die Enumeration „ApplicationPriority“ besteht aus einer sortierten Liste der einzelnen Prioritäten. Jedes Paar zweier benachbarter Prioritäten kann über den Relationsoperator in Formel 4-2 verglichen werden. Der Relationsoperator besitzt die Eigenschaft der Transitivität, weshalb jedes Paar zweier verschiedener Prioritäten miteinander verglichen werden kann. Damit können die Anwendungen einer Anwendungsdomäne (DomainArchitecture) in Bezug auf ihre jeweilige Priorität untereinander verglichen werden.

$$>: ApplicationPriority\ x\ ApplicationPriority \qquad (4\text{-}2)$$

Da Konkurrenz in Bezug auf Komponenten nur zwischen Anwendungen derselben Domäne auftritt, besitzt jede Anwendungsdomäne ihre eigene Sortierung ihrer Prioritäten. Die Gruppe von Anwendungskonfigurationen einer Domäne wurde in Kapitel 3 als Anwendungsdomänenkonfiguration definiert. Im Metamodell in Abbildung 4-2 wird eine Anwendungsdomänenkonfiguration als vereinigte Menge von Anwendungskonfigurationen von Anwendungsinstanzen dargestellt. Diese Anwendungsinstanzen sind alle über ihre jeweilige Anwendung (Application) über „referencedBy“ mit derselben Anwendungsdomäne (DomainArchitecture) verbunden.

Abbildung 4-3 zeigt die Beziehungen der Mengen Systemkonfiguration (SC), Anwendungsdomänenkonfiguration (ADC) und Anwendungskonfiguration (AC) zueinander. Eine Systemkonfiguration besteht aus einer Menge von Anwendungskonfigurationen, welche wieder zu Anwendungsdomänenkonfigurationen zusammengefasst werden können. Jede Anwendungskonfiguration kann genau in einer Anwendungsdomänenkonfiguration enthalten sein.

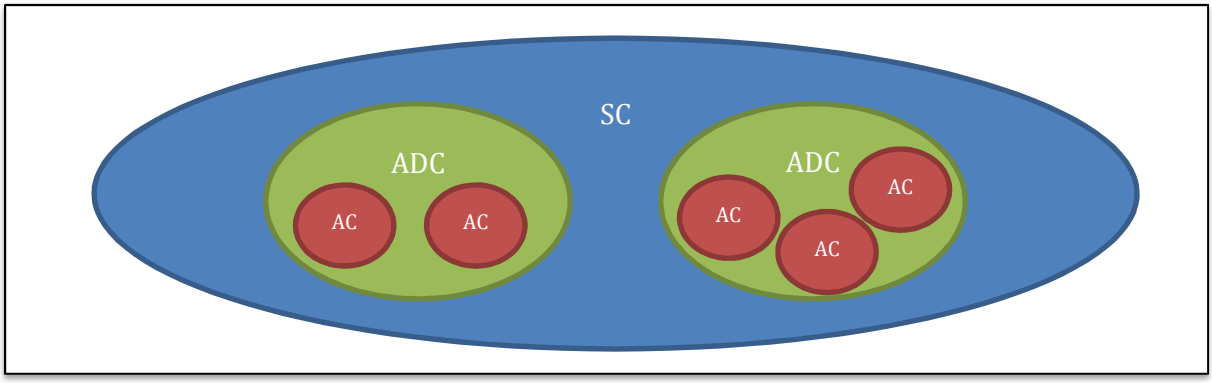

Abbildung 4-3: Mengenbeziehungen der Konfigurationen

4.1.2 Definition einer gültigen Systemkonfiguration

In Kapitel 3 wurde der Begriff Systemkonfiguration eingeführt und in Kapitel 4.1.1 an dem Metamodell in Abbildung 4-2 beschrieben. Jede gültige Systemkonfiguration soll nun die Bedingung der 5. funktionalen Anforderung erfüllen. Höher priorisierte Anwendungsinstanzen sollen immer vor niedriger priorisierten Anwendungsinstanzen verschaltet werden. Wenn bei einer Systemkonfiguration eine Anwendungsinstanz ohne Anwendungskonfiguration existiert, aber eine Anwendungskonfiguration mit freien und an niedriger priorisierten Anwendungsinstanzen gebundenen Komponenten gebildet werden kann, ist diese Systemkonfiguration ungültig. Um dies auch formal zu beschreiben, werden erst einmal einige Prädikate und Mengen definiert.

Das Prädikat „ApplicationIsRunning“ wurde schon in Kapitel 4.1.1 definiert. Nun wird noch ein Prädikat benötigt, um eine Aussage darüber treffen zu können, ob eine Anwendungsinstanz mit einer Menge von Komponenten laufen kann.

$$ApplicationCanRun: ApplicationInstance\ x\ P(DynamicAdaptiveComponent) \rightarrow Bool$$

$$ApplicationCanRun(app, DAC) \Leftrightarrow \exists dac \subseteq DAC: \big(app.contains.sellectedComponents = dac \wedge ApplicationIsRunning(app)\big) \quad (4\text{-}3)$$

Das Prädikat in Formel 4-3 erhält eine Anwendungsinstanz sowie eine Menge von Komponenten und prüft, ob die Anwendungsinstanz mit einer Teilmenge der Komponentenmenge laufen kann.

Für die Definition einer gültigen Systemkonfiguration werden noch zwei Mengen an Komponenten benötigt. Die eine Menge ist die Menge der freien Komponenten. Die andere Menge besteht aus der Menge an Komponenten, welche an Anwendungsinstanzen gebunden sind, bei der diese Anwendungsinstanzen unter der gegebenen Priorität liegen (siehe Formeln 4-4 und 4-5).

$$freeDAC = System.containedComponents \setminus \begin{pmatrix} System.containedInstances.contains.selectedComponents \\ \cup System.containedInstances.contains.assignedComponents \end{pmatrix} \quad (4\text{-}4)$$

$$\forall p \in ApplicationPriority: boundedDAC_p = \bigcup(\forall app \in ApplicationInstance | app.refersTo.priority < p: app.contains.selectedComponents) \quad (4\text{-}5)$$

Die Menge „freeDAC" entspricht der Menge der Komponenten, welche an keine Anwendungsinstanz gebunden sind. Die Menge „boundedDAC$_p$" entspricht dabei der Menge von Komponenten, welche über „selectedComponents" an die Anwendungsinstanzen gebunden sind, bei denen die Priorität der Anwendungsinstanz unter „p" liegt. Formel 4-6 fasst die Prädikate und Mengen nun zu einem Prädikat für gültige Systemkonfigurationen zusammen.

$$SystemIsValid: System \rightarrow Bool$$

$$SystemIsValid(s) \Leftrightarrow \forall app \in s.containedInstances: \neg ApplicationIsRunning(app) \Rightarrow \neg ApplicationCanRun\big(app, freeDAC \cup boundedDAC_{app.refersTo.priority}\big) \quad (4\text{-}6)$$

Mittels dieser Definition können aus der Menge an möglichen Systemkonfigurationen alle gültigen Systemkonfigurationen bestimmt werden. Im Folgenden werden nur noch gültige Systemkonfigurationen betrachtet.

4.1.3 Analyse der Systemkonfiguration

Da in einem System meist mehrere gültige Systemkonfigurationen erstellt werden können, muss nun die Systemkonfiguration ermittelt werden, welche als optimal angesehen wird. Zu diesem Zweck müssen die Systemkonfigurationen nach Kriterien sortiert werden, welche sich aus den Anforderungen aus Kapitel 3 ergeben.

Die Anforderungen in Kapitel 3 beziehen sich bei einer optimalen Systemkonfiguration ausschließlich auf die Gewichtung der einzelnen Anwendungskonfigurationen. In dieser Arbeit wird der Wert der Systemkonfiguration eines Systems als Summe der Werte der Anwendungskonfigurationen definiert (siehe Formel 4-7).

$$assessmentFunction: System \rightarrow N$$

$$assessmentFunction(s) = \sum s.containedInstances.weight \quad (4\text{-}7)$$

Eine Zielfunktion (ZF) soll nun die optimale Systemkonfiguration eines Systems durch Maximierung der Bewertungsfunktion angeben (siehe 4-8).

$$Zielfunktion: System \rightarrow System$$

$$\forall s, s2 \in System: Zielfunktion(s) = smax \wedge assessmentFunction(smax) \geq assessmentFunction(s2) \quad (4\text{-}8)$$

Die Zielfunktion ermittelt aus der Menge an gültigen Systemkonfigurationen diejenige, welche über die Funktion „assessmentFunction“ den höchsten Wert liefert. Mittels der Erweiterungen am Komponentenmodell und der Zielfunktion ist es nun möglich, die Systemkonfigurationen zu bewerten und daraus die optimale Systemkonfiguration zu ermitteln.

4.2 Dezentraler Algorithmus zur Bestimmung der optimalen Systemkonfiguration

In der zweiten Forschungsfrage wird nach einem dezentralen Algorithmus gefragt, welcher in der Lage ist, die optimale Systemkonfiguration eines Systems zu ermitteln. Dabei wird die eben erstellte Definition einer optimalen Systemkonfiguration verwendet. Ein Überblick zu diesem Algorithmus wird in diesem Abschnitt gegeben, in Kapitel 5 folgt anschließend eine ausführliche Beschreibung.

4.2.1 Definition der Akteure

Dezentrales Verhalten wird ebenfalls im Bereich der Multiagentensysteme verwendet, für die eine Menge von Verfahren zur Ressourcenoptimierung existiert. Multiagentensysteme bestehen aus rationalen Agenten, welche anhand ihrer Umgebung und ihrer Strategien versuchen, ihr jeweiliges Ziel zu erreichen (Wooldridge 2009). Damit die Aktionen der Agenten, welche nur auf ihr eigenes Ziel hinarbeiten, auch eine globale optimale Lösung anstreben, existiert für die Agenten ein an die jeweilige Situation angepasstes Belohnungssystem. Der Mechanismenentwurf (Russell und Norvig 2012) beschäftigt sich mit der Entwicklung solcher Systeme. In dieser Arbeit wird ein Agent als selbstständig agierende Einheit definiert, welcher als Ziel das Erreichen einer optimalen Systemkonfiguration besitzt.

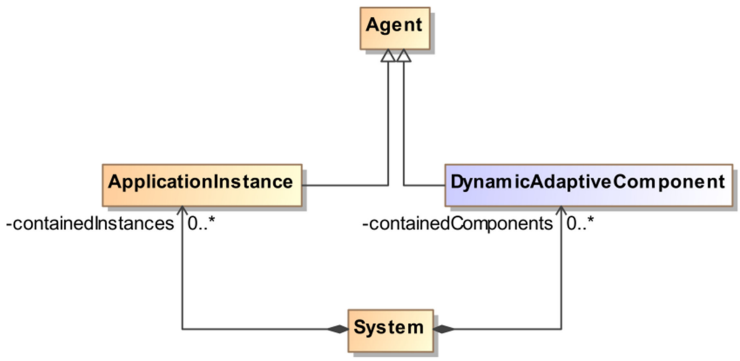

Abbildung 4-4: Darstellung der im System vorhandenen Akteure

Übertragen auf das in Kapitel 4 beschriebene System werden Anwendungsinstanzen und Komponenten als Agenten dargestellt. Im Folgenden werden diese aktiven Teilnehmer, wie in Abbil-

dung 4-4 dargestellt, **Akteure** (Agent) des Systems genannt. Jeder Akteur besitzt ein Ziel, welches er mittels einer Strategie zu erreichen versucht.

4.2.2 Ziele der Akteure

Wie in Abbildung 4-4 dargestellt, enthält das System 2 Gruppen von Akteuren. Eine Gruppe besteht aus den Anwendungsinstanzen, welche versuchen, Komponenten für eine Anwendungskonfiguration zu erhalten. Mittels dieser Anwendungskonfiguration besitzen die Anwendungsinstanzen dann jeweils eine Gewichtung. Die Gewichtungen werden von dem Architekten der Anwendungsdomäne gesetzt, welcher auch die Prioritäten und deren Reihenfolge festsetzt. Über die Gewichtungen und Prioritäten wird die in dieser Anwendungsdomäne erwartete optimale Anwendungsdomänenkonfiguration als Teil der optimalen Systemkonfiguration definiert.

Komponenten gehören zur 2. Gruppe von Akteuren und versuchen mit der maximalen Menge an Anwendungsinstanzen zugeordnet zu werden, um dadurch ebenfalls eine optimale Systemkonfiguration zu erreichen. Je mehr Anwendungskonfigurationen erstellt werden können, desto höher ist der Wert der Bewertungsfunktion der Systemkonfiguration.

4.2.3 Ein analytisches Verfahren zur Berechnung der optimalen Systemkonfiguration

In der Forschungsfrage wird außerdem das sichere Erreichen der optimalen Systemkonfiguration als Ziel genannt. Um immer die optimale Systemkonfiguration ermitteln zu können, sind analytische Verfahren notwendig. Da die Verschaltung von Komponenten zu Anwendungen aber ein NP-vollständiges Problem darstellt (Petty et al. 2003), wird ein analytisches Verfahren eine exponentielle Laufzeit in Abhängigkeit von der Anzahl der Komponenten besitzen.

Diese Arbeit beschäftigt sich mit der grundsätzlichen Bildung der optimalen Systemkonfiguration ohne Schwerpunkt auf eine optimale Laufzeit. Daher wird eine exponentielle Laufzeit bei Verwendung eines analytischen Verfahrens in dieser Arbeit akzeptiert. In weiteren Arbeiten kann die Verwendung von z. B. heuristischen Verfahren die Laufzeit bei der Suche nach einer akzeptablen Systemkonfiguration, welche nicht zwingend der optimalen Systemkonfiguration entsprechen muss, stark verringern.

4.2.4 Ablauf des Verfahrens

Das Verfahren zur Bestimmung einer optimalen Systemkonfiguration besteht aus fünf Schritten, welche in einer Schleife endlos wiederholt werden. Die fünf Schritte werden hier für einen ersten Überblick vorgestellt und in Kapitel 5 ausführlich beschrieben.

Der 1. Schritt des Verfahrens zum dezentralen Erstellen einer optimalen Systemkonfiguration liegt in dem Austausch von Informationen zwischen Komponenten und Anwendungsinstanzen. Denn zu Anfang besitzen Komponenten und Anwendungsinstanzen in einem offenen dynamischen System kein Wissen über andere Komponenten und Anwendungsinstanzen. Jeder Akteur benötigt für seine eigenen Entscheidungen kein vollständiges Wissen zu dem gesamten System. Daher werden nur die Informationen ausgetauscht, welche Anwendungsinstanzen und Komponenten auch benötigen.

Zu Anfang besitzen Anwendungsinstanzen keinen Überblick über die existierenden Komponenten im System. Daher senden sie in diesem Schritt ihre Spezifikationen für die benötigten Komponenten als offene Anfrage an das ganze System. Die Komponenten, welche die Anfrage erhalten, können nun prüfen, welche Spezifikation welcher Anwendungsinstanz sie erfüllen können, und senden ihre Antworten gezielt an die entsprechenden Anwendungsinstanzen zurück.

Damit die 5. funktionale Anforderung aus Kapitel 3 eingehalten wird und eine gültige Systemkonfiguration erzeugt werden kann, senden die Komponenten ihre Antworten nur den Anwendungsinstanzen zu, welche die höchste Priorität besitzen. Falls keine dieser Anwendungsinstanzen genügend Komponenten zusammenbekommt, um eine Anwendungskonfiguration erstellen zu können, senden die Komponenten ihre Antworten anschließend den Anwendungsinstanzen mit der nächstniedrigeren Priorität zu. Dies wiederholt sich so lange, bis mindestens eine Anwendungsinstanz eine Anwendungskonfiguration erstellen kann oder die Anwendungsinstanzen alle Prioritäten abgelehnt haben. Im Falle einer Ablehnung aller Anwendungsinstanzen, die geantwortet haben, warten die Komponenten auf neue Anfragen.

Eine Anwendungsinstanz kennt nach diesem Schritt alle potenziell passenden Komponenten, die dieser Anwendungsinstanz zugeordnet werden können, und kann im 2. Schritt alle möglichen Anwendungskonfigurationen für sich berechnen. Die Komponenten, welche den Anwendungsinstanzen zugesagt haben, werden bei der Verschaltung erst wieder in Schritt 5 aktiv.

Nach der Berechnung aller möglichen Anwendungskonfigurationen versuchen die Anwendungsinstanzen der jeweiligen Anwendungsdomäne in Schritt 3 Anwendungsdomänenkonfigurationen zu erstellen, siehe Definition in Kapitel 4.1. Da nur Anwendungsinstanzen einer Anwendungsdomäne untereinander um Komponenten konkurrieren, können die Anwendungsdomänenkonfigurationen der einzelnen Anwendungsdomänen voneinander unabhängig erstellt werden.

Allerdings benötigt die Anwendungsinstanz zur Erstellung der möglichen Anwendungsdomänenkonfigurationen die möglichen Anwendungskonfigurationen der übrigen Anwendungsinstanzen derselben Anwendungsdomäne. Daher tauschen diese Anwendungsinstanzen untereinander in Schritt 3 ihre Anwendungskonfigurationen aus und erstellen jede für sich mögliche Anwendungsdomänenkonfigurationen. Allerdings werden bei der Erstellung der Anwendungsdomänenkonfigurationen von jeder Anwendungsinstanz nur diejenigen erstellt, bei denen die jeweilige Anwendungsinstanz auch eine Anwendungskonfiguration beisteuert.

Da die Bedingungen für eine gültige Anwendungskonfiguration der einzelnen Anwendungsinstanzen aber nicht mitgesendet werden, erfüllen nicht alle erstellten Anwendungsdomänenkonfigurationen die Bedingungen aller beteiligten Anwendungsinstanzen. Durch das Zusammenfassen von Komponenten zweier Anwendungskonfigurationen ist es daher möglich, dass die Bedingungen mindestens einer Anwendungsinstanz dabei verletzt werden. Das Anwendungsbeispiel nennt einen Kunden und eine VIP-Anwendung als Beispiel dafür. Eine Kundenanwendungsinstanz besitzt keine Einschränkung, dass eine andere Anwendungsinstanz seine Roboterkomponente prinzipiell mit verwenden kann. Bei der VIP-Anwendungsinstanz wäre dann aber eine Bedingung verletzt. Da die Kundenanwendungsinstanz das aber nicht weiß, erstellt sie Anwendungsdomänenkonfigurationen, wo sie mit der VIP-Anwendungsinstanz zusammenarbeitet.

Diese ungültigen Anwendungsdomänenkonfigurationen werden nun im 4. Schritt ausgefiltert. Gleichzeitig können sich die beteiligten Anwendungsinstanzen auf eine der verbleibenden Anwendungsdomänenkonfigurationen einigen. Dabei soll die Anwendungsdomänenkonfiguration ausgewählt werden, welche die höchste Gewichtung (weight) als Summe der beteiligten Anwendungskonfigurationen besitzt. Allerdings ist es möglich, dass keine Anwendungsdomänenkonfiguration existiert, für die alle beteiligten Anwendungsinstanzen eine Anwendungskonfiguration liefern. In diesem Fall werden mehrere um die Menge der Komponenten konkurrierende Anwendungsdomänenkonfigurationen gewählt, für die aber jede Anwendungsinstanz ausschließ-

lich in genau einer dieser Anwendungsdomänenkonfiguration eine Anwendungskonfiguration beisteuert.

Zur Realisierung dieses Schrittes wird das Verfahren „Feilschen“ aus der Spieltheorie verwendet (Russell und Norvig 2012). In diesem Verfahren bieten die Anwendungsinstanzen den daran beteiligten Anwendungsinstanzen Anwendungsdomänenkonfigurationen an. Dabei wird immer die mögliche Anwendungsdomänenkonfiguration mit der höchsten Gewichtung verwendet. Die Anwendungsinstanzen, welche ein solches Angebot erhalten haben, prüfen, ob ihre Bedingungen darin verletzt werden und ob es mit ihrem aktuellen Angebot übereinstimmt. Wenn dies nicht der Fall ist, wird eine Ablehnung gesendet. Bei Ablehnung verwenden die jeweiligen Anwendungsinstanzen die mögliche Anwendungsdomänenkonfiguration mit der nächstniedrigeren Gewichtung. Dies wiederholt sich, bis sich alle schließlich auf eine Anwendungsdomänenkonfiguration bzw. auf eine Menge von in Bezug auf Anwendungsinstanzen unabhängigen Anwendungsdomänenkonfigurationen geeinigt haben.

Im letzten Schritt müssen aus den jeweils erstellten Anwendungsdomänenkonfigurationen der im System existierenden Anwendungsdomänen diejenigen ausgewählt werden, welche in der Summe die optimale Systemkonfiguration darstellen. Um aus den lokal optimalen Anwendungsdomänenkonfigurationen die global optimale Anwendungsdomänenkonfiguration jeder Anwendungsdomäne nun dezentral ermitteln zu können, wird das Auktionsverfahren aus der Spieltheorie verwendet (Russell und Norvig 2012).

Die Komponenten starten hierbei Auktionen, in denen sie ihre Dienste anbieten. Die Anwendungsinstanz, welche noch Komponenten benötigt, schickt diesen Komponenten nun ihre Anwendungsdomänenkonfiguration als Angebot zu. Jede Komponente wiederum wählt dann unter allen Angeboten die Anwendungsdomänenkonfiguration mit der höchsten Gewichtung aus. So kann die global optimale Anwendungsdomänenkonfiguration schließlich aus den lokal optimalen Anwendungsdomänenkonfigurationen ermittelt werden.

Dieses Auswahlverfahren wird von allen Komponenten angewendet. Daher erhält die Anwendungsdomänenkonfiguration mit der höchsten Gewichtung von jeder beteiligten Komponente eine Zusage und kann im Anschluss umgesetzt werden. Die gewählten Anwendungsdomänenkonfigurationen aus allen Anwendungsdomänen bilden zusammen schließlich die optimale Systemkonfiguration.

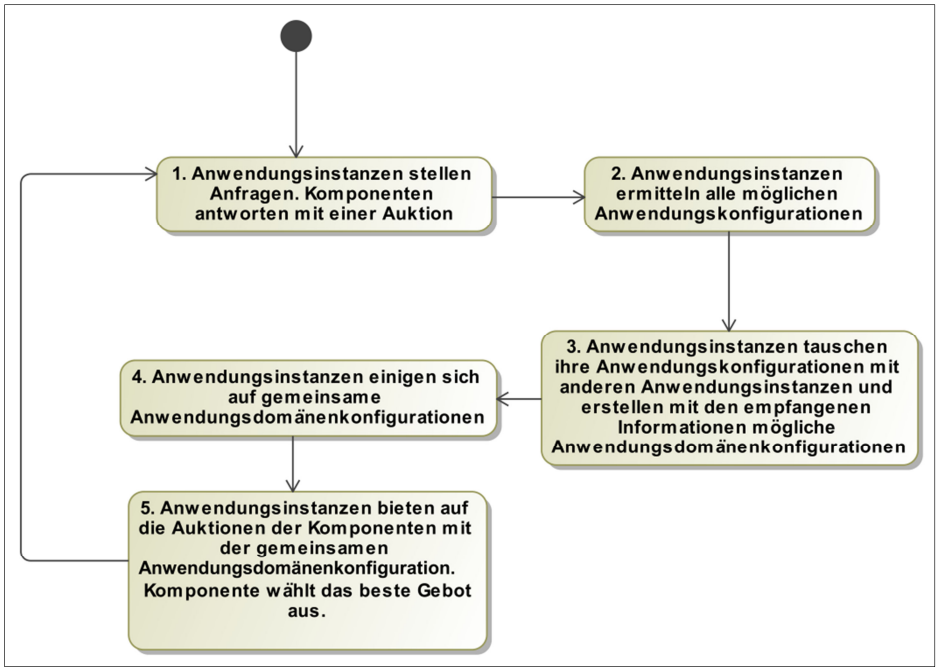

Abbildung 4-5: Ablauf des Lösungsverfahrens

Abbildung 4-5 fasst die beschriebenen Schritte in einem Ablaufdiagramm zusammen. Daraus wird deutlich, dass alle Akteure bei Schritt 1 beginnen. Außerdem besitzt der Ablauf eine Schleife, damit die Schritte wiederholt ausgeführt werden können. Der Algorithmus kann so in einem dynamischen System auf Änderungen reagieren.

4.3 Stabilität des dezentralen Algorithmus

Mit der 3. Forschungsfrage soll untersucht werden, wie die Stabilität des Algorithmus aus Kapitel 4.2 sichergestellt werden kann. Wie in Abbildung 4-5 dargestellt, wiederholen sich die Schritte des Algorithmus kontinuierlich. Dabei muss nun sichergestellt werden, dass eine einmal erstellte Systemkonfiguration beim nächsten Durchlauf nicht sofort wieder aufgelöst wird, um im schlimmsten Fall nicht exakt die gleiche Systemkonfiguration zu erstellen. Daher wird eine tatsächliche Neuverschaltung der Komponenten nur nach Schritt 5 vor Schritt 1 ausgeführt und zusätzlich nur, wenn eine bessere Systemkonfiguration gefunden wurde oder die bestehende Systemkonfiguration z. B. durch eine Prioritätsverletzung ungültig geworden ist.

Ein anderes Stabilitätsproblem kann auftreten, wenn eine Komponente oder Anwendungsinstanz immer wieder ausfällt und neu gestartet wird. Da sich damit die optimale Systemkonfiguration ggf. immer wieder ändern kann, wird dies eine ständige Neuverschaltung zur Folge haben. Eine Betrachtung der Anforderung aus Szenario 2 des Anwendungsbeispiels zeigt allerdings auf, dass eine bestehende Anwendungskonfiguration einer Anwendungsinstanz nicht von einer anderen Anwendungsinstanz gleicher Priorität, aber mit einer höheren Gewichtung aufgelöst werden soll. Damit kann eine nur kurz ausfallende Anwendungsinstanz oder Komponente bei Anwendungsinstanzen gleicher Priorität nachträglich keine Instabilität durch Neukonfiguration erzeugen. Die Ausnahme bilden Anwendungsinstanzen, welche eine höhere Priorität als die Priorität der verschalteten Anwendungsinstanzen besitzen. Da diese Anwendungsinstanzen Vorrang haben, kann eine so auftretende Instabilität nicht vom System her ohne zusätzliche Regeln oder einen manuellen Eingriff des Domänenarchitekten gelöst werden.

Als Folgerung der eben beschriebenen Lösungsideen zu dieser Forschungsfrage wurde das Verfahren aus Kapitel 4.2 dementsprechend angepasst. Die Akteure werden nach einem Durchlauf so lange in Schritt 1 verbleiben, bis sich am System, also an der Menge der gestarteten Komponenten und Anwendungsinstanzen etwas ändert. Anwendungsinstanzen, welche nach dem ersten Durchlauf keine Anwendungskonfiguration erstellen konnten, werden mit den übrig gebliebenen freien Komponenten ebenfalls keine Anwendungskonfiguration erstellen können. Sonst wäre dies bereits im vorherigen Schleifendurchlauf geschehen. Auch beim Betreten einer Anwendungsinstanz mit gleicher oder geringerer Priorität als die Anwendungsinstanzen der Anwendungsdomänenkonfiguration als Teil der Systemkonfiguration bleiben die Akteure in Schritt 1.

Zur Sicherstellung der Stabilität muss noch ein weiterer Punkt betrachtet werden, der sich mit den Systemkonfigurationen beschäftigt. Bis jetzt wurde angenommen, dass sich die Systemkonfigurationen über die Bewertungsfunktion eindeutig sortieren lassen und dass genau eine optimale Systemkonfiguration existiert. Da sich der Wert einer Systemkonfiguration aus der Summe der Anwendungsinstanzen mit erstellter Anwendungskonfiguration ergibt, ist es sehr wahrscheinlich, dass mehrere Systemkonfigurationen den gleichen Wert besitzen und dass sogar mehrere optimale Systemkonfigurationen mit gleichem maximalem Wert existieren. Damit determiniert der Algorithmus nicht mehr. Um dies aber gewährleisten zu können, werden Systemkonfigurationen mit gleichem Wert über einen eindeutigen Hashwert unterschieden, welcher sich aus der jeweiligen Struktur der Systemkonfiguration berechnet.

4.4 Evaluierung des Algorithmus

In der 4. und letzten Forschungsfrage wird die Frage gestellt, wie dieser Algorithmus auf Korrektheit in Bezug zur Problemstellung geprüft werden kann. Die Prüfung wird mittels des Evaluierens des Verfahrens an Beispielen durchgeführt.

Eine erste Prüfung findet in Kapitel 5 bei der Vorstellung des Lösungsverfahrens statt. Das Lösungsverfahren wird mit Szenario 4 aus dem Anwendungsbeispiel erläutert. Daran wird gezeigt, dass das Konzept der Lösung an einem Szenario exemplarisch ausgeführt werden kann und die erwartete Systemkonfiguration liefert.

Anschließend wird aus dem Lösungsverfahren in Kapitel 6 ein Prototyp entwickelt, welcher aus den Ausgangssituationen der einzelnen Szenarien des Anwendungsbeispiels automatisiert die jeweilige erwartete Systemkonfiguration bestimmen soll. Die Szenarien des Anwendungsbeispiels werden dann in Kapitel 7 einzeln ausgeführt und deren Systemkonfiguration wird mit der jeweiligen erwarteten Systemkonfiguration verglichen.

Abschließend werden noch einige weitere Szenarien entwickelt, anhand derer der Prototyp ebenfalls evaluiert wird. Damit soll gezeigt werden, dass das Lösungsverfahren nicht nur auf das Anwendungsbeispiel aus Kapitel 3 beschränkt ist.

4.5 Darstellung am Anwendungsbeispiel

Nach Erweiterung des Komponentenmodells in Abbildung 4-2 müssen nun die Anwendungen des Anwendungsbeispiels an das erweiterte Modell angepasst werden. Dafür wird ebenfalls eine Auflistung der für das Anwendungsbeispiel nötigen Prioritäten benötigt. Anschließend wird zu den einzelnen Szenarien der jeweilige Wert der daraus resultierenden Systemkonfiguration mittels der Bewertungsfunktion bestimmt.

4.5.1 Erweiterung der Anwendungen des Anwendungsbeispiels

Für die Erweiterung der Anwendungen des Anwendungsbeispiels müssen erst einmal die Prioritäten der zugehörigen Anwendungsdomäne, wie in Abbildung 4-6 abgebildet, aufgestellt werden. Dies wird für jede Anwendungsdomäne von dem zuständigen Domänenarchitekten durchgeführt.

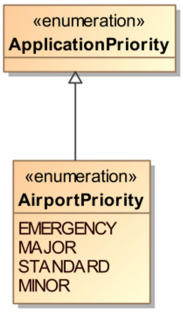

Abbildung 4-6: Prioritäten des Anwendungsbeispiels

Anwendungen mit der Priorität EMERGENCY besitzen in diesem Anwendungsbeispiel die höchste Priorität und werden vor Anwendung der Priorität MAJOR, STANDARD und MINOR verschaltet. Außer der Priorität wird den Anwendungen von dem Domänenarchitekten noch eine Gewichtung für jede Anwendungskonfiguration zugewiesen. Die Gewichtungen wurden so gewählt, dass die Zielsystemkonfigurationen aus den Szenarien des Anwendungsbeispiels daraus hergeleitet werden können. Sie bestehen bei allen Anwendungen, außer bei der Anwendung „Evakuierung“, aus einem konstanten Wert, weil diese Anwendungen nur Anwendungskonfigurationen bilden können, welche sich in ihrer Struktur gleichen. Bei der Evakuierungsanwendung hingegen können Anwendungskonfigurationen mit einer unterschiedlichen Anzahl von Roboterkomponenten erstellt werden. Daher hängt der Wert von der Anzahl der an diese Anwendung gebundenen Roboterkomponenten ab.

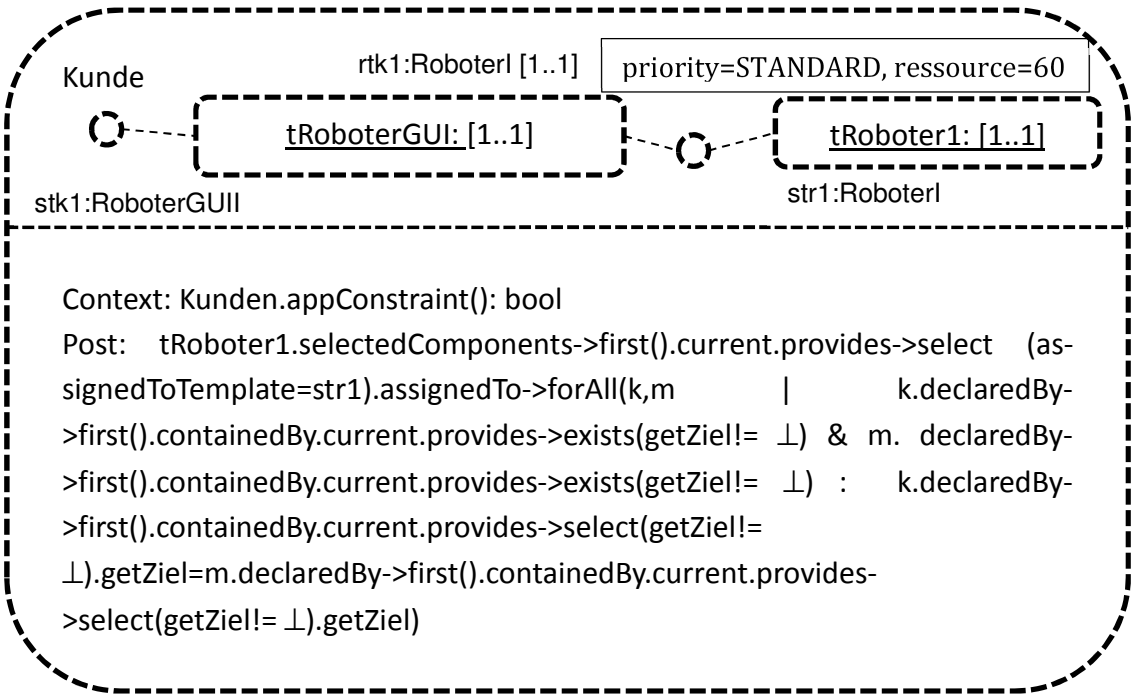

Abbildung 4-7: Erweiterte Anwendung Kunde

Der Anwendung „Kunde“ wird die Priorität STANDARD zugeordnet. Diese Priorität steht für den Normalbetrieb auf dem Flughafen, weshalb die meisten Anwendungen dieser Priorität zugeordnet sind. Die Gewichtung dieser Anwendung wurde auf 60 festgelegt. Dieser Wert ist genauso fiktiv wie die Anwendung selbst, weshalb die Höhe des Wertes keinen Bezug zu einem realen Wert besitzt. Allerdings ist die Differenz dieses Wertes zu den Werten der anderen Anwendungen des Anwendungsbeispiels von Bedeutung. Dies wird innerhalb dieses Kapitels durch die Szenarien verdeutlicht.

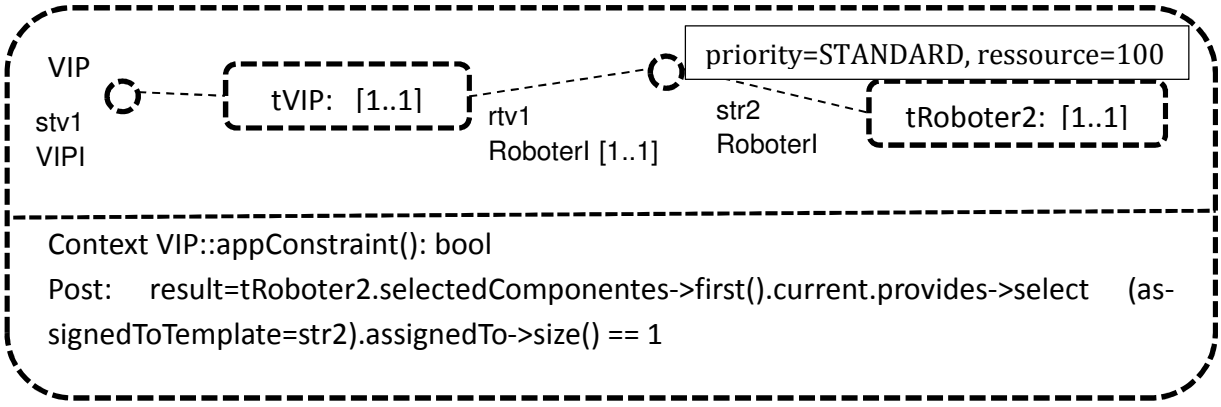

Abbildung 4-8: Erweiterte Anwendung VIP

Die Anwendung VIP erhält ebenfalls die Priorität STANDARD, da auch sie während des Normalbetriebs auf dem Flughafen ausgeführt werden soll. Allerdings erhält die Anwendung den Wert 100 als Gewichtung, welcher über dem Wert der Anwendung „Kunde“ liegt. Dies resultiert aus dem 1. Szenario des Anwendungsbeispiels, wo die Anwendung „VIP“ vor der Kundenanwendung verschaltet werden soll.

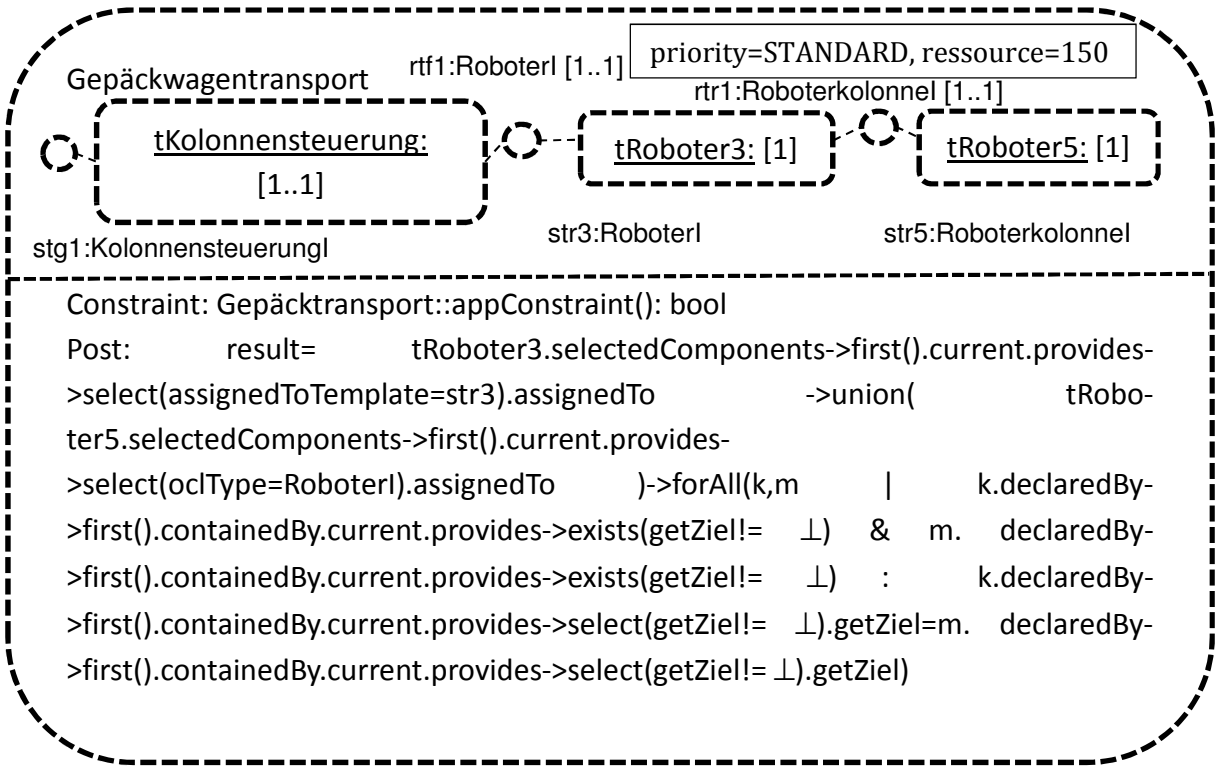

Abbildung 4-9: Erweiterte Anwendung Gepäckwagentransport

Wie die beiden eben beschriebenen Anwendungen soll auch die Gepäckwagentransportanwendung im Normalbetrieb auf dem Flughafen laufen, weshalb sie ebenfalls die Priorität STANDARD erhält. Um die Zielkonfiguration in Szenario 4 des Anwendungsbeispiels erreichen zu können, wurde die Gewichtung der Anwendung auf 150 gesetzt.

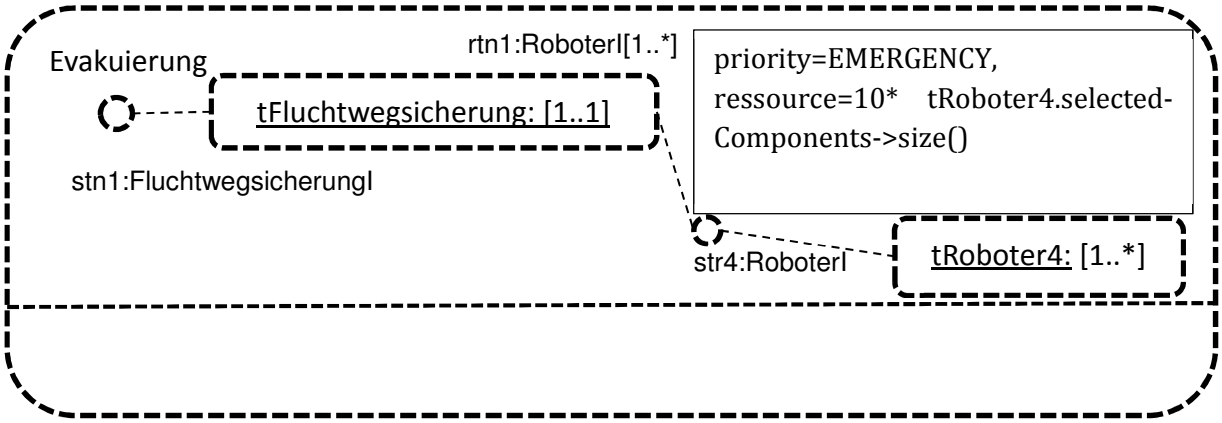

Abbildung 4-10: Erweiterte Anwendung Evakuierung

Der Evakuierungsanwendung wird als einzige Anwendung des Anwendungsbeispiels die Priorität EMERGENCY zugewiesen. Somit steht die Anwendung in ihrer Priorität, wie im Anwendungsbeispiel vorgesehen, über den der anderen drei Anwendungen des Anwendungsbeispiels. Die Gewichtung ist von der Anzahl der verbundenen Roboterkomponenten der erstellten Anwendungskonfiguration abhängig. Pro Roboterkomponente erhöht sie die Gewichtung um 10. Damit ist die optimale Anwendungskonfiguration also die mit der höchsten Gewichtung, die mit den meisten gebundenen Roboterkomponenten.

4.5.2 Erläuterung der Zielfunktion an dem Anwendungsbeispiel

Nachfolgend werden die Zielsystemkonfigurationen der einzelnen Szenarien des Anwendungsbeispiels dargestellt und in Bezug zu dem Wert der Bewertungsfunktion (assessmentFunction) aus Kapitel 4.1 gesetzt.

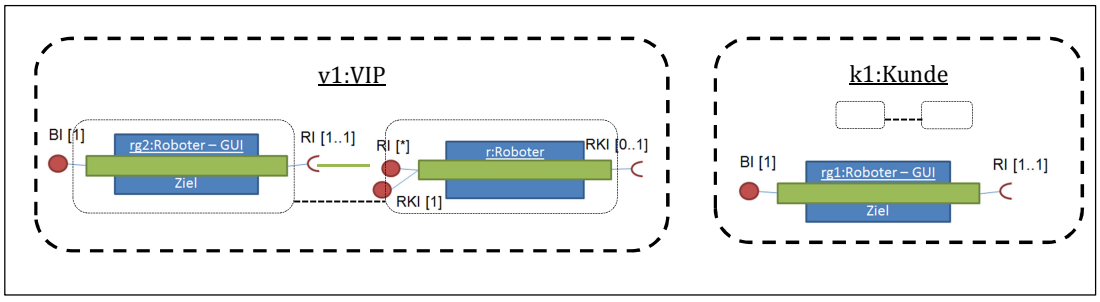

Abbildung 4-11: Zielsystemkonfiguration Szenario 1

In der erwarteten Zielsystemkonfiguration von Szenario 1 des Anwendungsbeispiels existiert nur die Anwendungskonfiguration der Anwendungsinstanz „v1“ der Anwendung „VIP“. Da diese Anwendungsinstanz laut Abbildung 4-8 für jede ihr zugeordnete Anwendungskonfiguration den Wert 100 als Gewichtung besitzt, ist dies auch der Wert, welchen die Bewertungsfunktion für die in diesem Szenario erwartete Systemkonfiguration berechnet. Wäre stattdessen der in diesem Szenario vorhandenen Anwendungsinstanz „k1“ der Anwendung „Kunde“ eine Anwendungskonfiguration zugeordnet, würde die Bewertungsfunktion nur den Wert 60 errechnen. Daher

stimmt die Aussage, dass die optimale Systemkonfiguration den höheren Wert aus der Bewertungsfunktion erhält, mit dem erwarteten Ergebnis aus diesem Szenario überein.

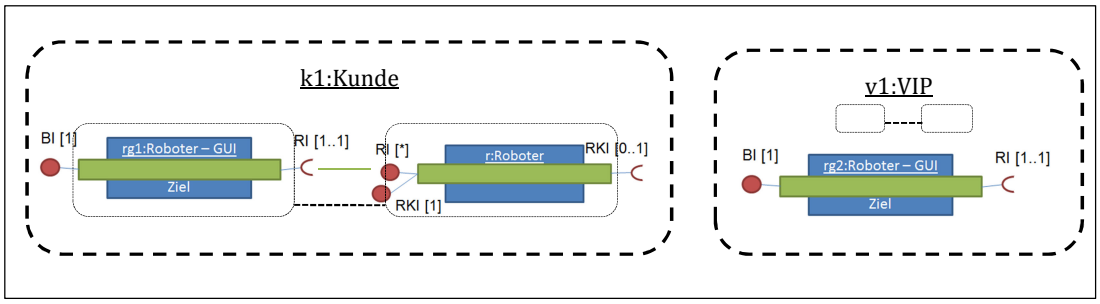

Abbildung 4-12: Zielsystemkonfiguration Szenario 2

Im 2. Szenario des Anwendungsbeispiels liefert die Bewertungsfunktion der erwarteten Zielsystemkonfiguration den Wert 60, wobei der bestmögliche Wert bei 100 liegt. Hier war allerdings die Anwendungsinstanz der Anwendung „Kunde“ schon mit der Roboterkomponente verbunden, bevor die Anwendungsinstanz der Anwendung „VIP“ mit gleicher Priorität gestartet wurde. Deshalb wurde die bestehende Anwendungskonfiguration der Kundenanwendungsinstanz hier laut den Anforderungen aus Szenario 2 nicht aufgelöst. Daraus resultiert, dass der Wert der erwarteten Zielsystemkonfiguration nicht immer dem höchsten Wert aller möglichen Systemkonfigurationen entspricht, sondern von Anwendungsinstanzen mit schon existierenden Anwendungskonfigurationen beeinflusst wird.

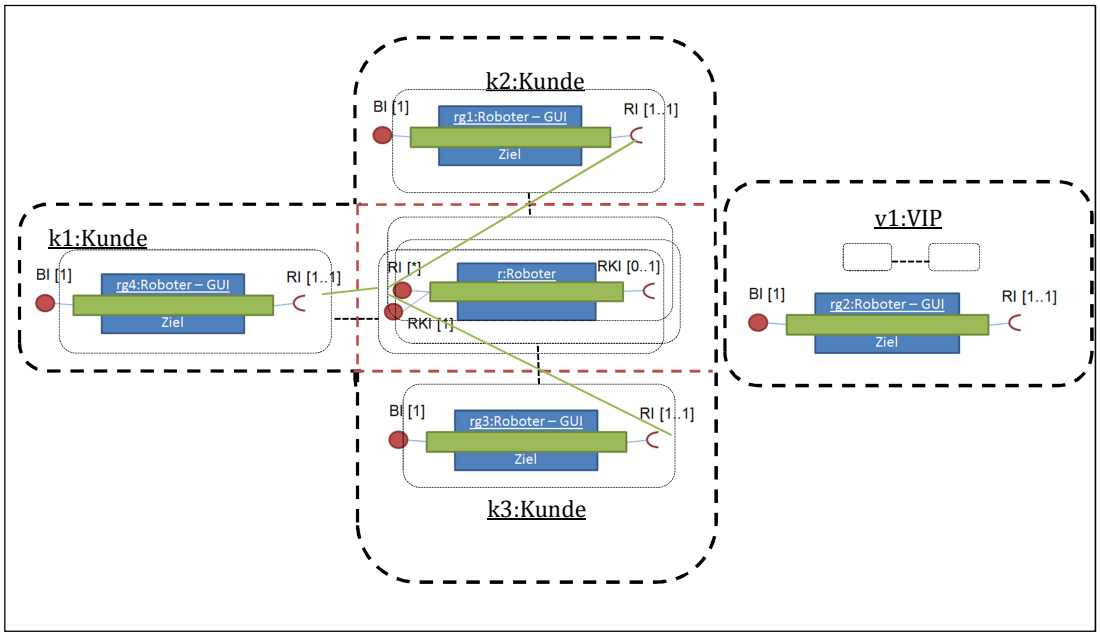

Abbildung 4-13: Zielsystemkonfiguration Szenario 3

Die Bewertungsfunktion liefert im 3. Szenario bei der erwarteten Zielsystemkonfiguration den Wert 180, da hier alle drei Kundenanwendungsinstanzen eine Anwendungskonfiguration besitzen. Die alternative Systemkonfiguration mit einer einzelnen Anwendungskonfiguration für die Anwendungsinstanz der Anwendung „VIP“ liefert mit dem Wert 100 aus der Bewertungsfunktion einen geringeren Wert. Wie in Szenario 1 ist hier wieder der höhere Wert ausschlaggebend.

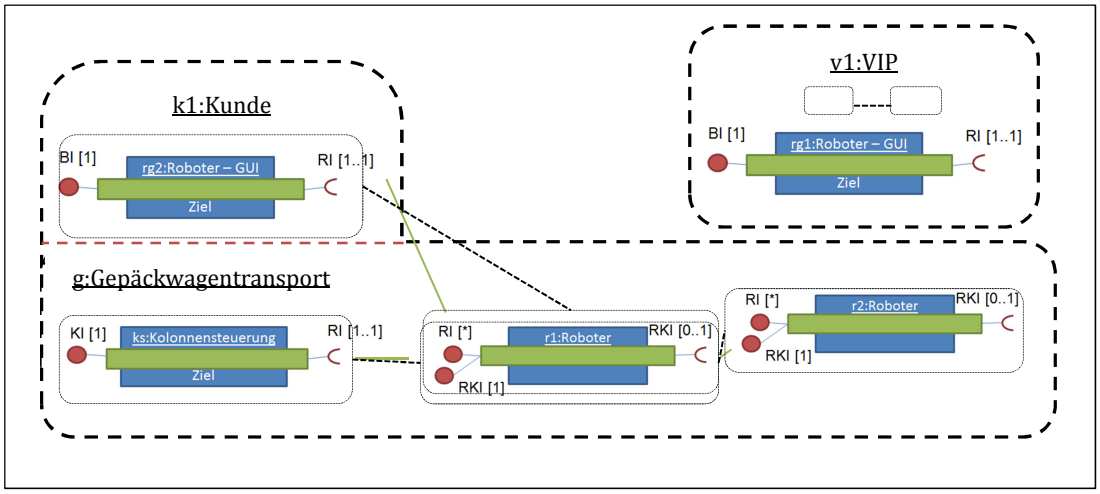

Abbildung 4-14: Zielsystemkonfiguration Szenario 4

Im 4. Szenario liefert die erwartete Zielsystemkonfiguration über die Bewertungsfunktion mit dem Wert 210 ebenfalls den höchstmöglichen erreichbaren Wert dieses Szenarios. Eine Systemkonfiguration mit den Anwendungskonfigurationen der Anwendungsinstanzen der Anwendungen „Kunde“ und „VIP“ liefert nur den Wert 160.

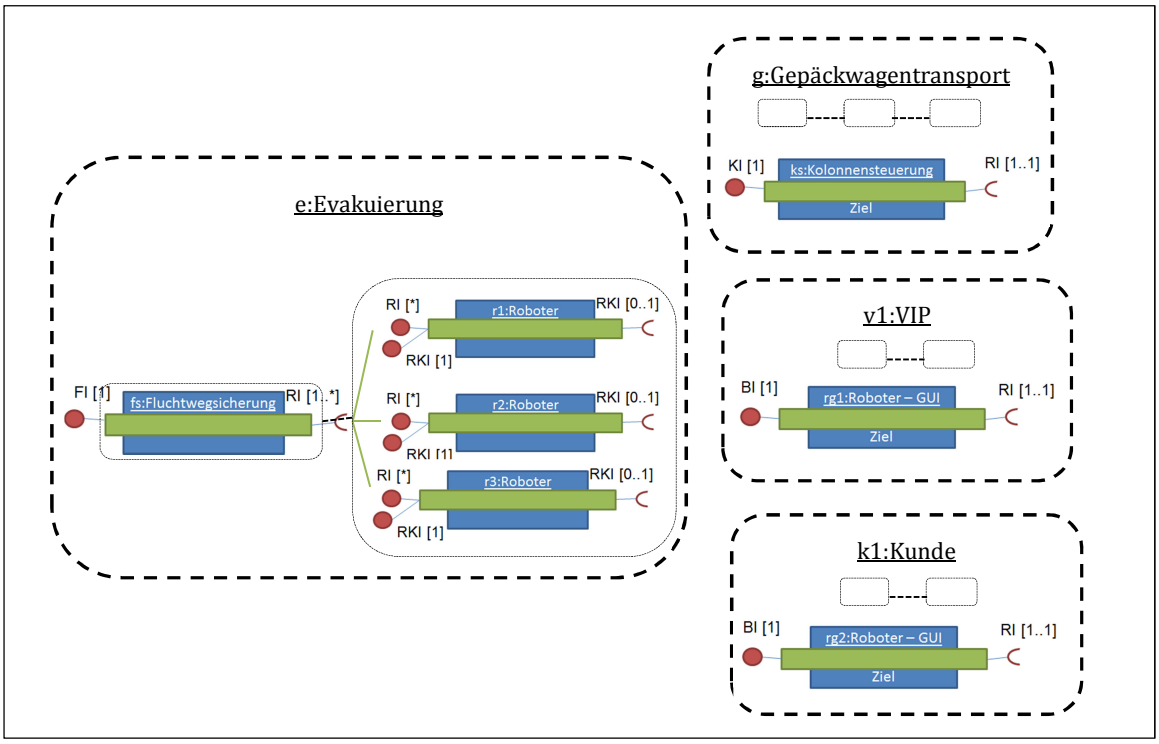

Abbildung 4-15: Zielsystemkonfiguration Szenario 5

Die erwartete Zielsystemkonfiguration im 5. und letzten Szenario des Anwendungsbeispiels liefert über die Bewertungsfunktion nur den Wert 30, welcher niedriger als jede andere mögliche Systemkonfiguration ist. Allerdings besitzt die Anwendungsinstanz, welche die einzige Anwendungskonfiguration für die erwartete gültige Zielsystemkonfiguration liefert, eine höhere Priorität als die übrigen im System vorhandenen Anwendungsinstanzen. Daher sind alle anderen Systemkonfigurationen ungültig, weil die Evakuierungsanwendung in diesem Szenario gegenüber den übrigen Anwendungsinstanzen im System bevorzugt behandelt werden muss.

Dieses Kapitel hat die Problemstellung analysiert, indem es zu jeder aufgestellten Forschungsfrage eine Antwort oder einen Überblick über die Antwort geliefert hat. Diese werden in den nachfolgenden Kapiteln noch weiter erläutert. Das nächste Kapitel beschäftigt sich mit der ausführlichen Beschreibung des Verfahrens, welches in der Lage ist, die Systemkonfigurationen mit den beschriebenen Anforderungen zu erstellen.

5 Lösungsverfahren

Das vorausgehende Kapitel hat die Problemstellung aus Kapitel 3 analysiert und Lösungen bzw. Lösungsansätze zu den einzelnen Forschungsfragen präsentiert. Dabei wurde das in (Klus 2013) beschriebene Komponentendiagramm dahin gehend erweitert, eine Systemkonfiguration beschreiben und bewerten zu können. Dieses Kapitel beschreibt ein Lösungsverfahren zur Erstellung der optimalen Systemkonfiguration unter Berücksichtigung der Anforderungen aus Kapitel 3 und der Bewertungsfunktion aus dem vorherigen Kapitel. Zuerst werden in diesem Kapitel die einzelnen Zustände der Akteure des Systems beschrieben. Anschließend werden die Zustände der Akteure in Bezug zu den einzelnen Schritten des Lösungsverfahrens gesetzt, welche in Abbildung 4-5 dargestellt sind. Im Anschluss wird das Lösungsverfahren anhand eines Szenarios des Anwendungsbeispiels aus Kapitel 3 verdeutlicht.

5.1 Verhalten der Akteure im System

Das Verhalten der Akteure wird mittels Zuständen strukturiert, welche jeweils einem Schritt des Lösungsverfahrens zugeordnet sind. Jeder Akteur kann sich jeweils ausschließlich in einem Zustand befinden. Er kommuniziert nur mit Akteuren des gleichen Typs, welche sich im gleichen Zustand befinden, oder mit Akteuren eines anderen Typs, welche sich in einem entsprechenden Zustand befinden. Anfragen von Akteuren anderer Zustände werden ignoriert bzw. abgelehnt. Als Akteure wurden im System Anwendungsinstanzen und Komponenten identifiziert, siehe Abbildung 4-4. Der Zustandsautomat jedes Akteurs wird nacheinander kurz im Überblick vorgestellt und dessen Zustände werden beschrieben.

5.1.1 Zustandsautomat der Komponenten

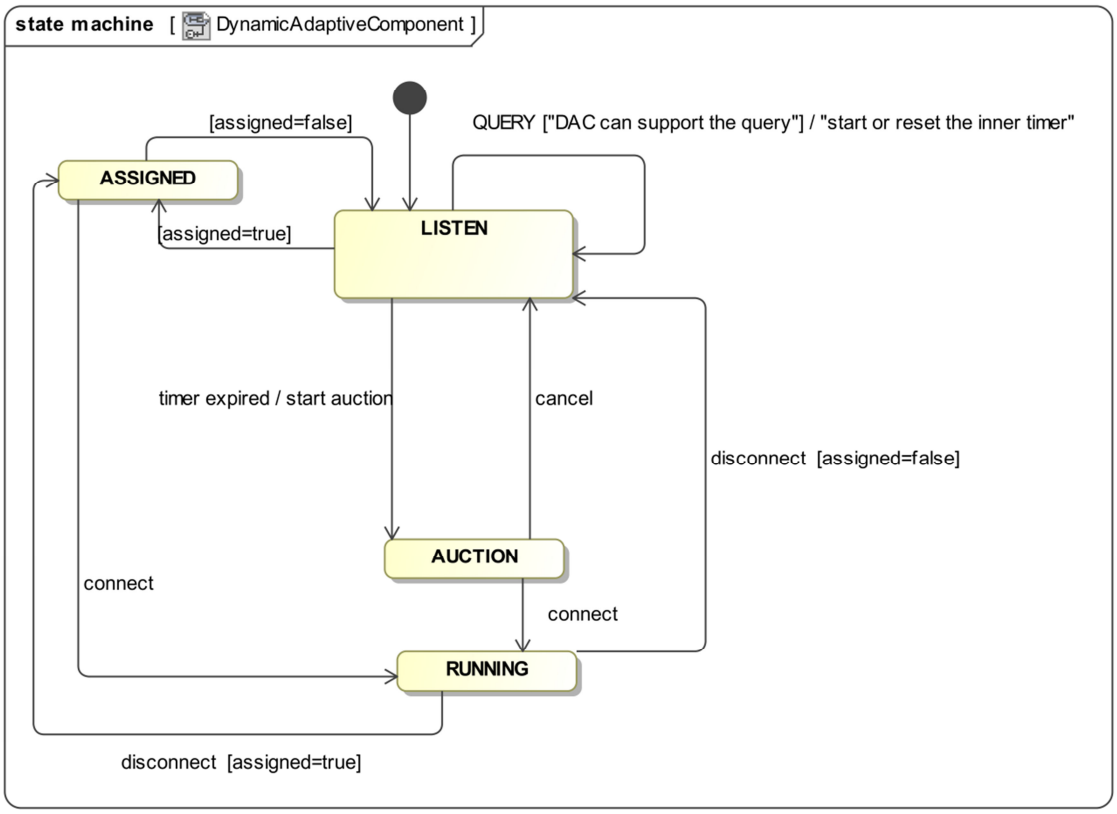

Abbildung 5-1: Zustandsautomat der Komponenten

In dem Zustandsdiagramm in Abbildung 5-1 sind die 4 Zustände der Komponenten und ihre Zustandsübergänge im Überblick dargestellt. Jede Komponente startet im Zustand LISTEN. Wenn eine Komponente einer Anwendungsinstanz fest zugeordnet ist, wird das Attribut „assigned“ der Komponente auf „true“ gesetzt und die Komponente wechselt sofort in den Zustand ASSIGNED. Im Zustand LISTEN warten die Komponenten auf mindestens eine Anfrage („query“) von einer Anwendungsinstanz. Bei der ersten Anfrage, welche die Komponente auch erfüllen kann, wird ein Timer gestartet. Bei jeder nachfolgenden passenden Anfrage wird der Timer zurückgesetzt. Wenn der Timer abgelaufen ist, wechselt die Komponente in den Zustand AUCTION. Die Komponente bleibt im Zustand AUCTION oder im Zustand ASSIGNED, bis sie den Aufruf zur Verschaltung („connect“) erhält, und wechselt anschließend in den Zustand RUNNING. Im Zustand RUNNING verbleibt die Komponente, bis sie sich über „disconnect“ aus der Verschaltung löst und je nach Inhalt des Attributs „assigned“ in den Zustand LISTEN oder ASSIGNED übergeht. Wenn die Komponente sich im Zustand AUCTION befindet und das Signal „cancel“ mit einer definierten Häufigkeit erhält, wechselt sie ebenfalls wieder in den Zustand LISTEN. Nachfolgend werden die Aufgaben der einzelnen Zustände genauer beschrieben.

5.1.1.1 Assigned

In diesem Zustand befindet sich die Komponente, wenn sie einer Anwendung fest zugeordnet ist, aber noch nicht läuft. Sie verbleibt in dem Zustand, bis sie von der zugeordneten Anwendungsinstanz die Aufforderung zur Verschaltung erhält. Anschließend verschaltet sich die Komponente mit anderen Komponenten entsprechend der Struktur der Anwendung und wechselt in den Zustand RUNNING.

5.1.1.2 Listen

Komponenten besitzen den Startzustand LISTEN, wenn sie keiner Anwendung zugeordnet sind. Diese Komponenten befinden sich dabei im ersten Schritt des Verfahrens, siehe Abbildung 4-5. Komponenten im Zustand LISTEN, welche eine Anfrage von einer Anwendungsinstanz erhalten, prüfen, ob sie die Anforderungen mindestens einer Schablone der Anwendungsbeschreibung der Anwendungsinstanz erfüllen. Wenn dies der Fall ist, startet die Komponente einen Timer und sendet der anfragenden Anwendungsinstanz, bei welcher die Komponente eine passende Schablone entdeckt hat, eine Antwort. Diese Antwort beinhaltet die Identifikationen der Schablonen, welche die Komponente aus ihrer Sicht belegen kann. Wenn bei einer Antwort an eine Anwendungsinstanz von der Komponente schon ein Timer gestartet wurde, wird dieser Timer zurückgesetzt. Nach Ablauf des Timers sendet die Komponente an alle Anwendungsinstanzen, denen sie geantwortet hat, ein „auction started“-Signal. Anschließend wechselt sie in den Zustand AUCTION.

5.1.1.3 Auction

Wenn sich eine Komponente in diesem Zustand befindet, wurden erfolgreich Anwendungen gefunden, welche diese Komponente nutzen möchten. Dieser Zustand wird dem fünften Schritt des Lösungsverfahrens zugeordnet, siehe Abbildung 4-5. In diesem Zustand starten die Komponenten jeweils eine Auktion und warten auf Angebote von allen Anwendungsinstanzen, für welche die Auktion gestartet wurde. Die Angebote beinhaltet eine Anwendungsdomänenkonfiguration und den Wert, bestehend aus dem Wert der Bewertungsfunktion und dem Hashwert der Struktur der Anwendungsdomänenkonfiguration. Nach Erhalt der Angebote aller Anwendungsinstanzen im Zustand LISTEN, die positiv geantwortet haben, wählen die Komponenten das Angebot mit dem höchsten Wert aus. Anschließend senden sie den Anwendungsinstanzen aus dem gewählten Angebot das Signal „connect“ und den übrigen Anwendungsinstanzen der Auktion das Signal „cancel“. Nach dem Erhalt des Signals „connect“ von den Anwendungsinstanzen be-

ginnen die Komponenten mit der Verschaltung und wechseln bei erfolgreicher Verschaltung in den Zustand RUNNING. Bei Erhalt des Signals „cancel“ von einer Anwendungsinstanz wird diese aus der Menge der an der Auktion beteiligten Anwendungsinstanzen entfernt. Wenn alle Anwendungsinstanzen hingegen jeweils das Signal „cancel“ gesendet haben, wechselt die Komponente wieder in den Zustand LISTEN zum ersten Schritt des Lösungsverfahrens.

5.1.1.4 Running

Der Zustand RUNNING kann keinem der fünf Schritte des Lösungsverfahrens zugeordnet werden, weil das Lösungsverfahren diese Komponente nun erfolgreich mit einer Menge von Anwendungen verschaltet hat. In diesem Zustand führen die Komponenten ihre jeweiligen anwendungsbezogenen Aufgaben aus. Komponenten verbleiben in diesem Zustand, bis sie entweder von allen Anwendungsinstanzen nicht mehr benötigt oder zwecks Erstellung einer neuen Systemkonfiguration neu verschaltet werden. In dem Fall, dass die Komponente nicht mehr benötigt wird, wechselt sie wieder in den Zustand LISTEN und befindet sich dann wieder im ersten Schritt des Lösungsverfahrens.

5.1.2 Zustandsautomat der Anwendungsinstanzen

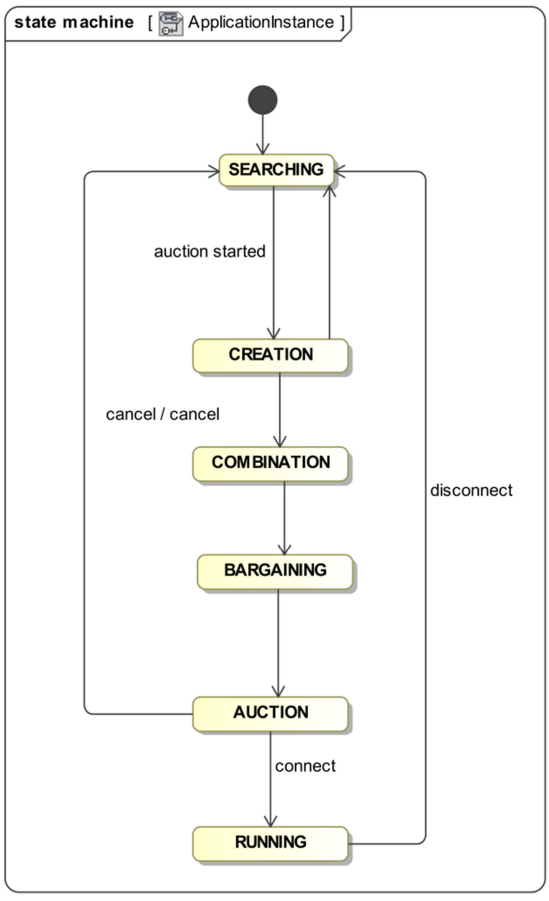

Abbildung 5-2: Zustandsautomat der Anwendungsinstanzen

In Abbildung 5-2 werden die Zustände der Anwendungsinstanzen und die bestehenden Zustandsübergänge im Überblick dargestellt. Die Anwendungsinstanzen beginnen in dem Zustand SEARCHING und wechseln anschließend in den Zustand CREATION zur Erstellung der möglichen Anwendungskonfigurationen, wenn sie genügend Signale „auction started“ von einer ausreichenden Zahl an Komponenten erhalten haben. Vom Zustand CREATION kann die Anwendungsinstanz zurück zum Zustand SEARCHING gelangen, wenn keine Anwendungskonfiguration er-

stellt werden konnte, wobei sie den zugehörigen Komponenten das „cancel"-Signal sendet. Bei mindestens einer erstellten Anwendungskonfiguration wechselt die Anwendungsinstanz in den Zustand COMBINATION. Anschließend wechselt die Anwendungsinstanz nach Erstellung aller Anwendungsdomänenkonfigurationen vom Zustand COMBINATION in den Zustand BARGAINING und nach Abarbeitung des Zustands in den Zustand AUCTION. Mit dem Erhalt des Signals „cancel" startet die Anwendungsinstanz den Zyklus wieder im Zustand SEARCHING neu. Über das Signal „connect" wechselt die Anwendungsinstanz schließlich in den Zustand RUNNING. In diesem Zustand verbleibt die Anwendungsinstanz, bis sie über das Signal „disconnect" für ihren nächsten Durchlauf in den Zustand SEARCHING wechselt.

5.1.2.1 Searching

Anwendungsinstanzen starten im Zustand SEARCHING. Mit diesem Zustand befinden sie sich in Schritt 1 des Lösungsverfahrens, siehe Abbildung 4-5. Anwendungsinstanzen in dem Zustand benötigen für die weiteren Schritte des Lösungsverfahrens eine Menge an Komponenten. Daher schicken sie an alle Komponenten im System jeweils eine Nachricht. Die Nachricht enthält alle Schablonen der Anwendungsinstanz, für die eine Komponente benötigt wird. Anschließend warten die Anwendungsinstanzen auf Antworten von Komponenten. Die Anwendungsinstanzen bleiben im Zustand SEARCHING, bis sie von allen Komponenten, welche auf eine Anfrage positiv geantwortet haben, das „auction started"-Signal erhalten haben. Wenn alle nötigen Schablonen mindestens von der minimal benötigten Menge an Komponenten eine positive Antwort erhalten haben, wechselt die jeweilige Anwendungsinstanz in den Zustand CREATION. Wenn mindestens eine nötige Schablone keine Antwort erhalten hat, existiert eine Anwendungskonfiguration der Anwendungsinstanz. In dem Fall sendet sie allen Komponenten, die geantwortet haben, das Signal „cancel", bleibt im Zustand SEARCHING und wiederholt ihre Anfrage. Die übrigen Anwendungsinstanzen wechseln mit dem Zustand CREATION zum nächsten Schritt des Verfahrens.

5.1.2.2 Creation

In diesem Zustand befinden sich die Anwendungsinstanzen im zweiten Schritt des Lösungsverfahrens. Ziel des zweiten Schrittes ist die Ermittlung aller möglichen Anwendungskonfigurationen aus den erhaltenen Antworten der Komponenten. Um jede mögliche Anwendungskonfiguration zu finden, wird in dem Lösungsverfahren ein simpler Brute-Force-Algorithmus verwendet. Das Suchen nach allen möglichen Anwendungskonfigurationen ist notwendig, um im Anschluss daran die optimale Systemkonfiguration ermitteln zu können. Anschließend wechselt die Anwendung in den Zustand COMBINATION. Konnte keine Anwendungskonfiguration erstellt werden, wechselt die Anwendungsinstanz für den nächsten Zyklus wieder in den Zustand SEARCHING und sendet das Signal „cancel" an die beteiligten Komponenten. Außerdem löscht sie sich aus der Clique.

5.1.2.3 Combination

Im Zustand COMBINATION führen die Anwendungen den dritten Schritt des Lösungsverfahrens aus, siehe Abbildung 4-5. Das Ziel dieses Schrittes besteht in der Erstellung aller möglichen Anwendungsdomänenkonfigurationen. Dafür ermitteln alle Anwendungsinstanzen alle möglichen Konkurrenten um die Komponenten und fassen diese in einer Clique zusammen. Danach tauschen die Anwendungen Informationen innerhalb der Clique aus. Jede Anwendungsinstanz einer Clique sendet ihre in dem Zustand CREATION erstellten Anwendungskonfigurationen zusammen mit dem entsprechenden Wert aus der Bewertungsfunktion an alle anderen Teilnehmer ihrer Clique. Die Anwendungsbeschreibung wird dabei allerdings nicht gesendet. Daher können die einzelnen Anwendungsinstanzen nach der anschließenden Gruppierung unterschiedliche Anwendungsdomänenkonfigurationen erhalten. Jede Anwendungsinstanz kann ihre Anwen-

dungsdomänenkonfigurationen nur auf die Einhaltung ihrer eigenen Bedingungen testen. Das erzeugt Anwendungsdomänenkonfigurationen, für welche die beteiligten Anwendungsinstanzen womöglich nicht alle ihre Bedingungen erfüllt sehen, sich aber der Berechnungsaufwand des Prüfens einer Anwendungsbeschreibung auf die einzelnen Anwendungsinstanzen verteilt. Das Herausfiltern gültiger Anwendungsdomänenkonfigurationen wird im Zustand BARGAINING vorgenommen.

Vor dem Wechsel in den Zustand BARGAINING wird die Liste der Anwendungsdomänenkonfigurationen nach der Größe des zugehörigen Wertes über die Bewertungsfunktion aus Kapitel 4 absteigend sortiert. Wenn zwei Elemente der Liste den gleichen Wert besitzen, wird bei beiden ein Hashwert über die enthaltenen Anwendungskonfigurationen gebildet. Das Element mit dem höheren Hashwert kommt vor dem Element mit dem geringeren Hashwert. Somit ist sichergestellt, dass jede Liste aller Mitglieder der Clique dieselbe Reihenfolge bei Anwendungsdomänenkonfigurationen mit gleichem Wert besitzt. Eine strikte Ordnung der Anwendungsdomänenkonfigurationen ist für den nächsten Zustand von Bedeutung.

5.1.2.4 Bargaining

In diesem Zustand führen die Anwendungen den vierten Schritt des Lösungsverfahrens aus. Darin soll aus allen erstellten Anwendungsdomänenkonfigurationen eine einzelne oder eine Menge von Anwendungsdomänenkonfigurationen ausgewählt werden, welche im fünften Schritt den Komponenten angeboten werden sollen. Alle Anwendungsinstanzen, welche sich im Zustand BARGAINING befinden, warten darauf, dass die übrigen Anwendungsinstanzen ihrer Clique ebenfalls diesen Zustand erreichen. Danach beginnen alle Anwendungsinstanzen mit dem Feilschen. Jede Anwendungsinstanz wählt das erste Element ihrer Liste und schickt es anschließend als Angebot an alle Anwendungsinstanzen, welche eine Anwendungskonfiguration zu dieser Anwendungsdomänenkonfiguration liefern. Zu diesem Angebot gehören auch der Ressourcenwert aus der Bewertungsfunktion und der Hashwert über die Struktur der Konfiguration. Wenn hingegen keine weitere Anwendungsinstanz beteiligt ist, die Menge also nur aus einer Anwendungskonfiguration besteht, wählt die Anwendungsinstanz dieses Element als die für sich beste Lösung. Danach wechselt sie in den Zustand AUCTION.

Angebote von Anwendungsinstanzen werden in allen Zuständen außer im Zustand BARGAINING abgelehnt. Die übrigen Anwendungsinstanzen im Zustand BARGAINING reagieren auf Angebote anderer Anwendungsinstanzen wie folgt:

- Wenn der Wert des Angebots, bestehend aus dem Ressourcenwert und dem Hashwert, den Wert des ersten Elements der Liste übersteigt, erfüllt das Angebot nicht alle Bedingungen der Anwendungsinstanz. Daher wird als Antwort eine Ablehnung gesendet. Ansonsten würde das Angebot sich in der Liste vor dem aktuell ersten Element befinden.
- Wenn der Wert dem Wert des ersten Elements entspricht, wurde also bei beiden dieselbe Anwendungsdomänenkonfiguration gewählt. Daher sendet die Anwendungsinstanz auf das Angebot als Antwort eine Zustimmung. Ohne eindeutige Sortierung der Anwendungsdomänenkonfigurationen mittels eines Hashwerts kann diese Folgerung nicht getroffen werden.
- Wenn der Wert unter dem Wert des ersten Elements liegt, kann das Angebot in einer späteren Iteration von Bedeutung sein. Daher merkt sich die Anwendungsinstanz dieses Angebot und wartet mit der Antwort. Außerdem werden Angebote aus der eigenen Liste gelöscht, welche einen höheren Wert als das aus dem Angebot besitzen und bei denen die anfragende Anwendungsinstanz eine Anwendungskonfiguration liefert. Diese An-

wendungskonfigurationen stehen nicht in der Liste der anfragenden Anwendungsinstanz. Daher können sich die Teilnehmer der Clique auch nicht auf diese einigen, weil die anfragende Anwendungsinstanz diese Konfiguration ablehnen wird.

Je nach erhaltenen Antworten auf ihre Angebote reagieren die Anwendungsinstanzen folgendermaßen:

- Bei einer Ablehnung eines gesendeten Angebots wird das erste Element aus der Liste gelöscht und eine neue Iteration mit dem neuen ersten Element der Liste gestartet. Zusätzlich werden die gemerkten Angebote erneut daraufhin geprüft, ob andere Anwendungsinstanzen für dieses Angebot schon das gleiche Angebot gesendet haben.
- Wenn dem Angebot von allen beteiligten Anwendungen zugestimmt wurde, wird diese Gruppe von Anwendungskonfigurationen gewählt und der Zustand in AUCTION gewechselt. Die Liste der gemerkten Angebote dieser Anwendungsinstanz wird dabei abgelehnt.

Da jede Anwendungsinstanz in ihrer jeweiligen Liste auch die eigenen Anwendungskonfigurationen besitzt, wird in diesem Schritt jede Anwendungsinstanz immer ein Element der aus dieser Liste auswählen können. Eine Anwendungskonfiguration wird dabei als Anwendungsdomänenkonfiguration mit nur einem Teilnehmer betrachtet. So ist sichergestellt, dass alle Anwendungsinstanzen der Clique immer eine Anwendungsdomänenkonfiguration auswählen. In diesem Zustand wurden nun ungültige Anwendungsdomänenkonfigurationen herausgefiltert und Anwendungsdomänenkonfigurationen als lokales Optimum für jede Anwendungsinstanz ermittelt. Aus diesen müssen nun noch die Anwendungsdomänenkonfigurationen gesucht werden, welche anschließend die optimale Systemkonfiguration darstellen.

5.1.2.5 Auction

In diesem Zustand wird der 5. Schritt des Lösungsverfahrens mit dem Ziel der Ermittlung der optimalen Anwendungsdomänenkonfiguration ausgeführt. Anwendungsinstanzen senden jeweils ihre aus dem vorherigen Zustand gewählte Anwendungsdomänenkonfiguration als Angebot auf die Auktion der jeweiligen beteiligten Komponenten. Das Angebot beinhaltet ebenfalls den Ressourcenwert der Bewertungsfunktion aus Kapitel 4. Nach Erhalt des Signals „connect“ von allen Komponenten sendet die jeweilige Anwendungsinstanz ihrerseits den Komponenten das Signal „connect“ und wechselt in den Zustand RUNNING. Bei Erhalt des Signals „cancel“ hingegen sendet die Anwendungsinstanz den Komponenten, von denen die Anwendungsinstanz schon das Signal „connect“ erhalten hat, das Signal „cancel“. Anschließend beginnt die Anwendungsinstanz wieder im Zustand SEARCHING mit dem ersten Schritt des Lösungsverfahrens.

5.1.2.6 Running

Der Zustand RUNNING kann analog zum entsprechenden Zustand einer Komponente ebenfalls keinem der fünf Schritte des Lösungsverfahrens zugeordnet werden. Im Zustand RUNNING wird die eigentliche Anwendung, welche sich hinter der Anwendungsbeschreibung verbirgt, ausgeführt. Die Anwendungsinstanz überwacht die Komponenten, um auf den Ausfall von Komponenten entsprechend reagieren zu können. Wenn z. B. eine notwendige Komponente ausgefallen ist, wird die Anwendungskonfiguration aufgelöst und im Zustand SEARCHING mit dem ersten Schritt des Lösungsverfahrens nach einer neuen Anwendungskonfiguration gesucht.

5.2 Ablauf des Lösungsverfahrens

In diesem Abschnitt werden die Zustände aus der Sicht der fünf Schritte aus Abbildung 4-5 betrachtet. Im nachfolgenden Sequenzdiagramm ist ein exemplarischer Ablauf des Lösungsverfah-

rens an einer konkreten Menge von Komponenten und Anwendungen dargestellt. Die einzelnen Schritte des Lösungsverfahrens werden dabei durch die farbliche Markierung des Hintergrundes im Diagramm dargestellt.

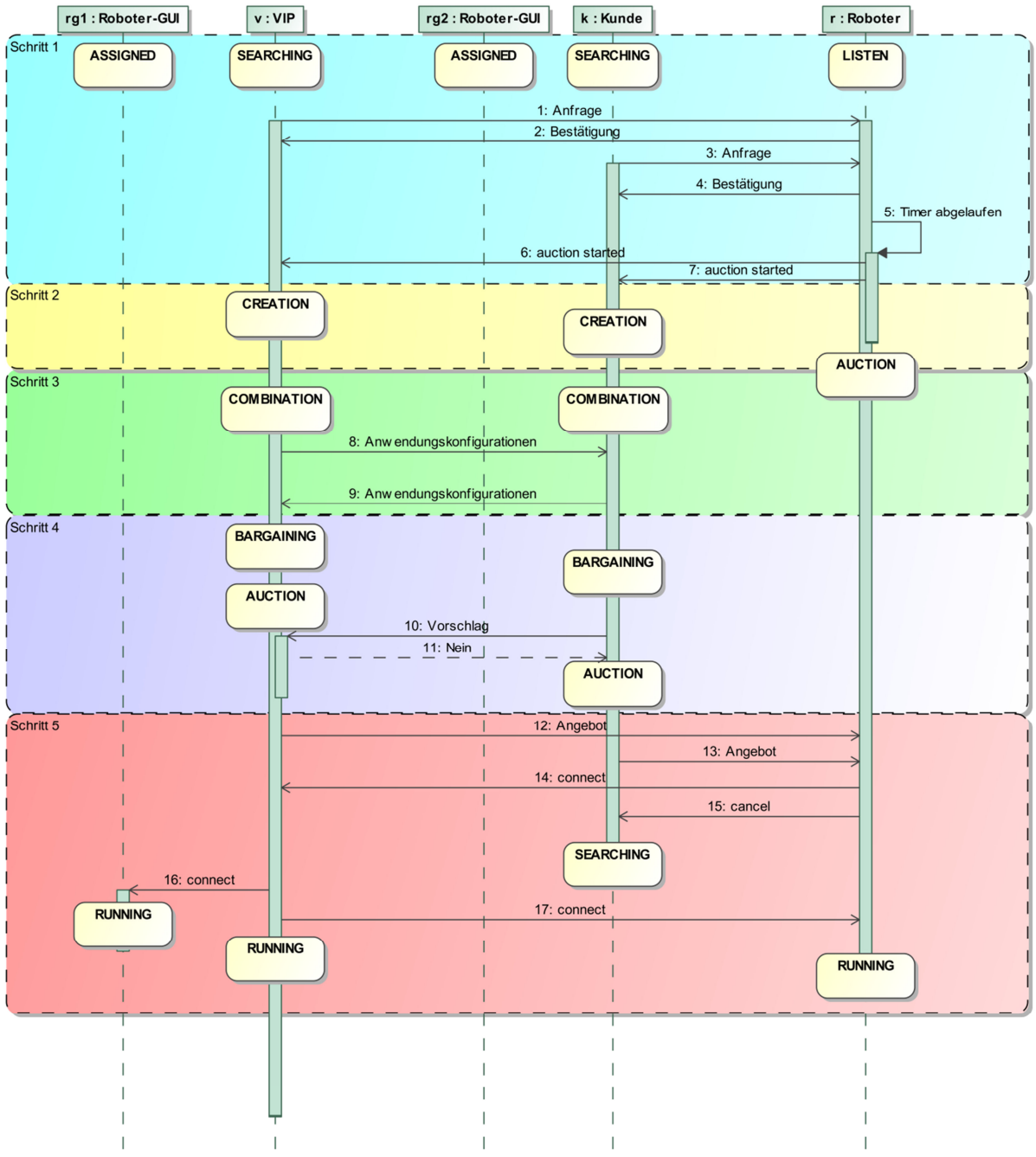

Abbildung 5-3: Zustandsübergänge anhand von Szenario 1 des Anwendungsbeispiels

Im Sequenzdiagramm in Abbildung 5-3 wird der Ablauf des Verfahrens an dem 1. Szenario des Anwendungsbeispiels aus Kapitel 3 dargestellt. Die beiden Anwendungsinstanzen der Anwendungen „VIP“ und „Kunde“ versuchen jeweils, mit der einzigen Roboterkomponente verbunden zu werden.

Im 1. Schritt des Verfahrens befinden sich die Anwendungsinstanzen im Zustand SEARCHING und die Komponenten in den Zuständen LISTEN oder ASSIGNED, abhängig davon, ob sie einer Anwendungsinstanz zugeordnet sind. Beide Anwendungsinstanzen stellen eine Anfrage, in der

sie nach einer Komponente suchen, welche die Anforderungen der jeweiligen Schablone erfüllen. Die Roboterkomponente prüft, ob sie die Anforderungen der jeweiligen Schablone erfüllen kann. Da dies der Fall ist, sendet sie an beide Anwendungsinstanzen eine Bestätigung. Über diese Bestätigung kennen sich nun beide Anwendungsinstanzen als Konkurrenten. Nachdem der Timer der Roboterkomponente abgelaufen ist, sendet sie an beide Anwendungsinstanzen das Signal „auction started“ und wechselt in den Zustand AUCTION. Beide Anwendungsinstanzen wechseln anschließend in den Zustand CREATION.

Im 2. Schritt befinden sich beide beteiligten Anwendungsinstanzen im Zustand CREATION und alle beteiligten Komponenten im Zustand AUCTION oder ASSIGNED. Die beiden Anwendungsinstanzen ermitteln nun alle möglichen Anwendungskonfigurationen und wechseln anschließend für Schritt 3 in den Zustand COMBINATION.

Im 3. Schritt senden sich beide Anwendungsinstanzen ihre Anwendungskonfigurationen zu und ermitteln daraus die möglichen Anwendungsdomänenkonfigurationen, wo sie jeweils eine Anwendungskonfiguration stellen. Die VIP-Anwendungsinstanz erhält nur eine Anwendungsdomänenkonfiguration mit ausschließlich ihrer Anwendungskonfiguration. Die Kundenanwendungsinstanz hingegen erstellt 2 mögliche Anwendungsdomänenkonfigurationen, eine mit nur ihrer Anwendungskonfiguration und eine mit zusätzlich noch der Anwendungskonfiguration der VIP-Anwendungsinstanz. Anschließend wechseln beide Anwendungsinstanzen in den Zustand BARGAINING.

Im 4. Schritt wählen beide Anwendungsinstanzen jeweils ihre Anwendungsdomänenkonfiguration mit dem höchsten Wert, welche die Bewertungsfunktion aus Kapitel 4 liefert. Da die VIP-Anwendungsinstanz nur eine Anwendungsdomänenkonfiguration besitzt, an der keine weitere Anwendungsinstanz beteiligt ist, wählt sie diese aus und wechselt in den Zustand AUCTION. Die Kundenanwendungsinstanz wählt zuerst die Anwendungsdomänenkonfiguration aus, an der die VIP-Anwendungsinstanz beteiligt ist, und sendet sie als Anfrage an die VIP-Anwendungsinstanz. Diese lehnt die Anfrage ab, da sie sich schon in einem anderen Zustand befindet. Darauf wählt die Kundenanwendungsinstanz die zweite und letzte Anwendungsdomänenkonfiguration aus, an der keine weiteren Anwendungsinstanzen beteiligt sind, und wechselt ebenfalls in den Zustand AUCTION.

Beide Anwendungsinstanzen senden nun im 5. Schritt ihre Anwendungsdomänenkonfigurationen und den jeweiligen Wert an die Roboterkomponente. Diese wartet, bis sie beide Angebote erhalten hat, und wählt das Angebot der VIP-Anwendungsinstanz wegen des höheren Wertes aus. Anschließend sendet sie der VIP-Anwendungsinstanz das Signal „connect“ und der Kundenanwendungsinstanz das Signal „cancel“. Die Kundenanwendungsinstanz wechselt darauf für Schritt 1 wieder in den Zustand SEARCHING. Die VIP-Anwendungsinstanz sendet an die Komponenten Roboter und Roboter-GUI das „connect“-Signal und wechselt in den Zustand RUNNING. Beide angesprochenen Komponenten verschalten sich, wobei die Roboterkomponente an die VIP-Anwendungsinstanz gebunden wird. Anschließend wechseln beide Komponenten in den Zustand RUNNING.

5.3 Verfahren anhand des Anwendungsbeispiels verdeutlichen

Zur näheren Erläuterung wird das Verfahren nun auf ein komplexeres Szenario des Anwendungsbeispiels angewendet. Von den Szenarien des Anwendungsbeispiels wird Szenario 4 verwendet.

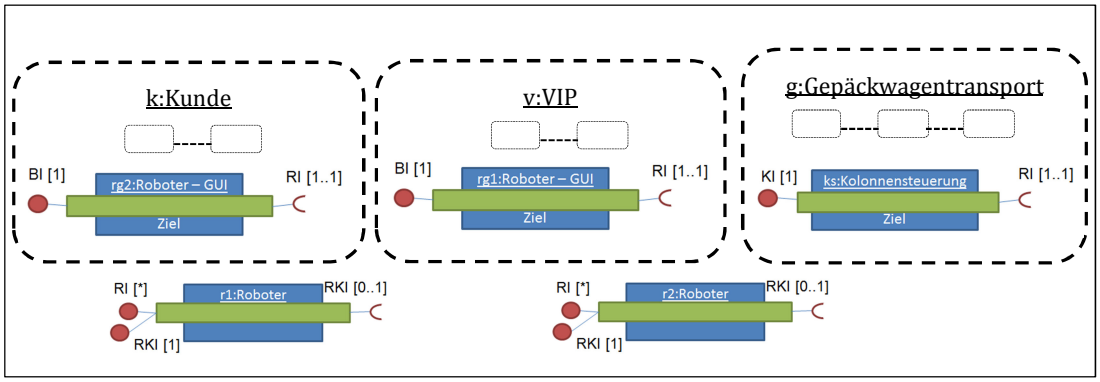

Abbildung 5-4: Szenario 4 in der Ausgangssystemkonfiguration

Schritt 1 des Verfahrens beginnt mit der Systemkonfiguration aus Abbildung 5-4. Die Komponente „rg1“ vom Typ „Roboter – GUI“ ist der Anwendungsinstanz „v“ der Anwendung „VIP“ explizit zugeordnet, die Komponente „rg2“ der Anwendungsinstanz „k“ und die Komponente „ks“ der Anwendungsinstanz „g“. Alle Komponenten und Anwendungen befinden sich jeweils in ihren Anfangszuständen. Bei den Anwendungsinstanzen ist dies der Zustand SEARCHING und bei den Komponenten der Zustand LISTEN. Die Komponenten „rg1“ und „rg2“ befinden sich im Zustand ASSIGNED.

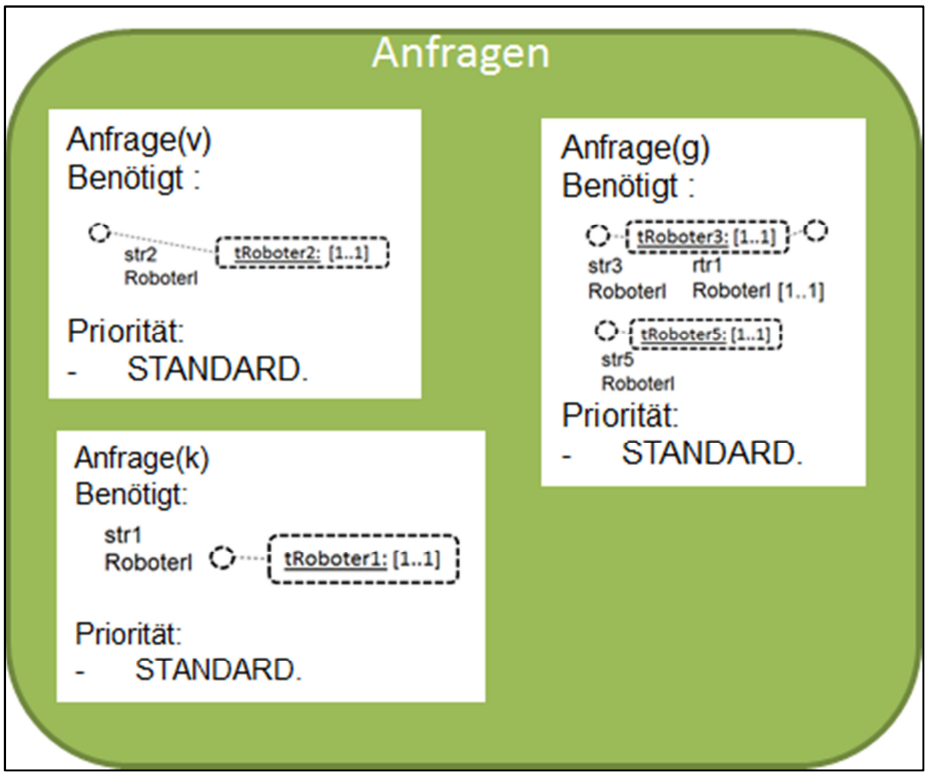

Abbildung 5-5: Darstellung aller Anfragen

Wie in Abbildung 5-5 dargestellt, stellen die drei Anwendungsinstanzen jeweils eine Anfrage. Jede Anfrage beinhaltet die Priorität und die Schablonen, für die Komponenten benötigt werden. Da ein Teil der Schablonen schon mit der benötigten Menge an Komponenten belegt ist, tauchen diese in der Anfrage nicht auf.

Anschließend antworten die Komponenten auf die Anfragen. Die beiden Roboterkomponenten antworten auf alle drei Anwendungsinstanzen und bei der Anwendungsinstanz „g“ sogar auf

beide Schablonen. Abbildung 5-6 zeigt alle gesendeten Antworten. Für die anschließende Cliquenbildung werden dabei die Anwendungen, die bis dahin geantwortet haben, als Liste gespeichert. Die Anwendungsinstanzen der Clique werden in der Abbildung in der Komponente dargestellt.

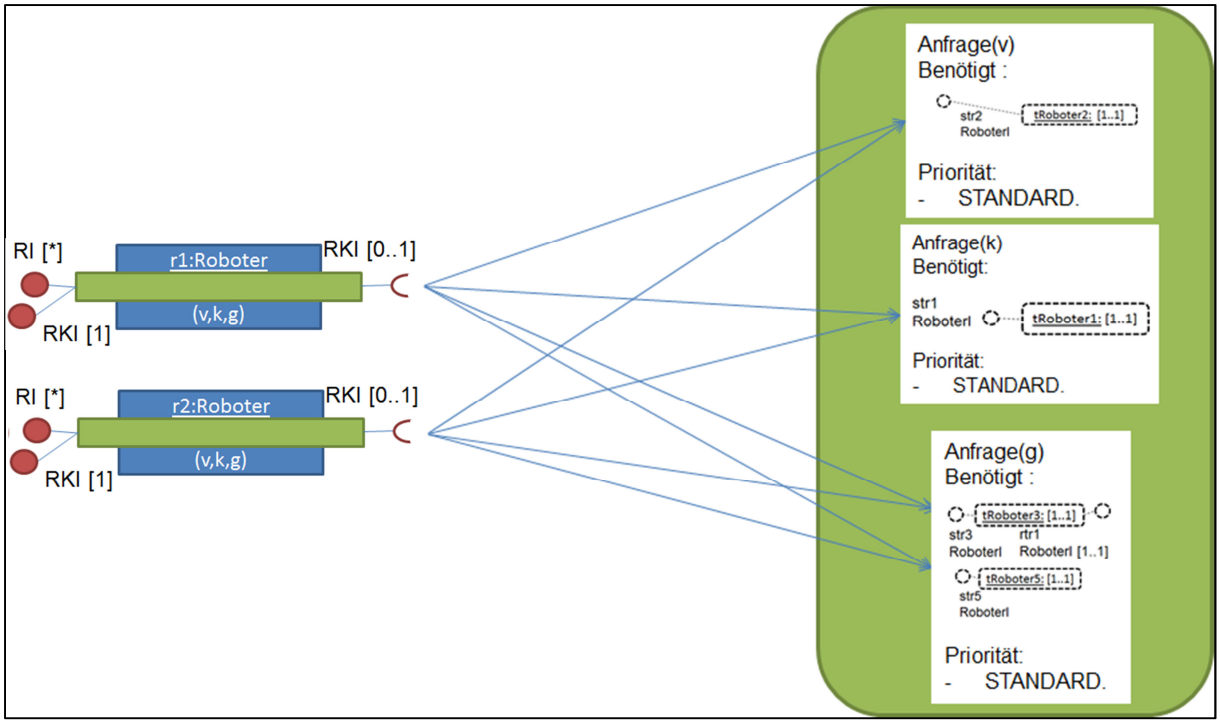

Abbildung 5-6: Antworten auf die Anfragen

Da im System keine weiteren Anwendungsinstanzen gestartet werden, senden alle nicht zugeordneten Komponenten das „auction started“-Signal und wechseln in den Zustand AUCTION. Die drei Anwendungen wechseln daraufhin in den Zustand CREATION.

In diesem Zustand erzeugen die drei Anwendungsinstanzen mit den Komponenten, die geantwortet haben, alle möglichen Anwendungskonfigurationen, siehe Abbildung 5-7. Anschließend wechseln alle Anwendungsinstanzen in den Zustand COMBINATION.

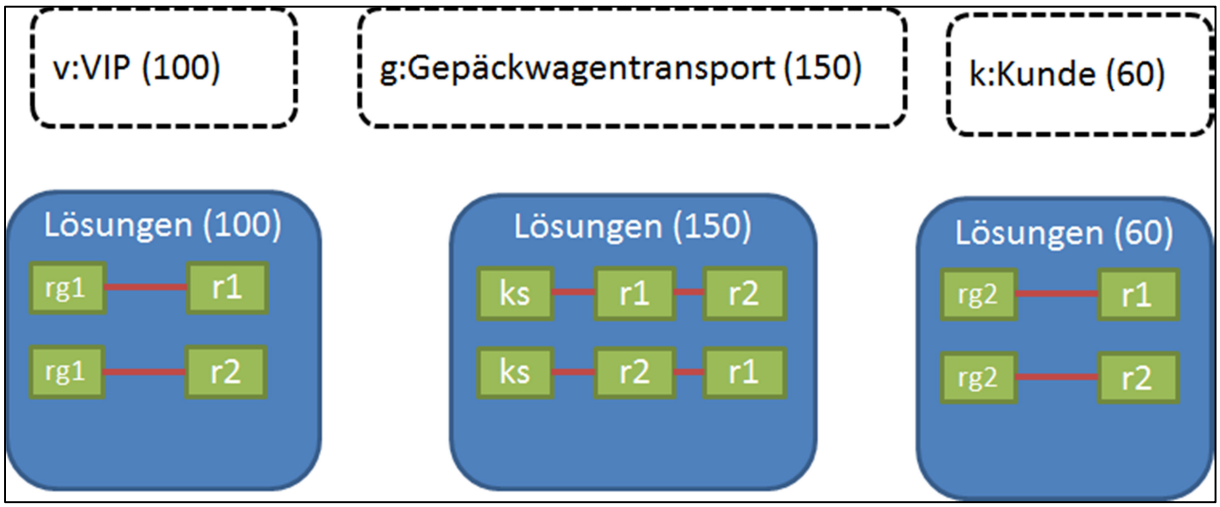

Abbildung 5-7: Liste aller möglichen Verschaltungen der Anwendungen

Nun ermitteln alle Anwendungsinstanzen ihre Clique und senden ihre möglichen Anwendungskonfigurationen an alle Anwendungsinstanzen in ihrer Clique. Die anschließende Erstellung der möglichen Anwendungsdomänenkonfigurationen, in denen die jeweilige Anwendungsinstanz eine Anwendungskonfiguration enthält, liefert für jede Anwendungsinstanz eine sortierte Liste an möglichen Anwendungsdomänenkonfigurationen, wie in Abbildung 5-8 dargestellt. Die Listen sind nach dem Wert sortiert, den die Bewertungsfunktion für die jeweilige Anwendungsdomänenkonfiguration liefert. Bei gleichen Werten zwischen zwei Konfigurationen entscheidet der Hashwert der Verschaltung über die Reihenfolge. Nach Erstellung der eigenen Liste wechselt die jeweilige Anwendungsinstanz in den Zustand BARGAINING.

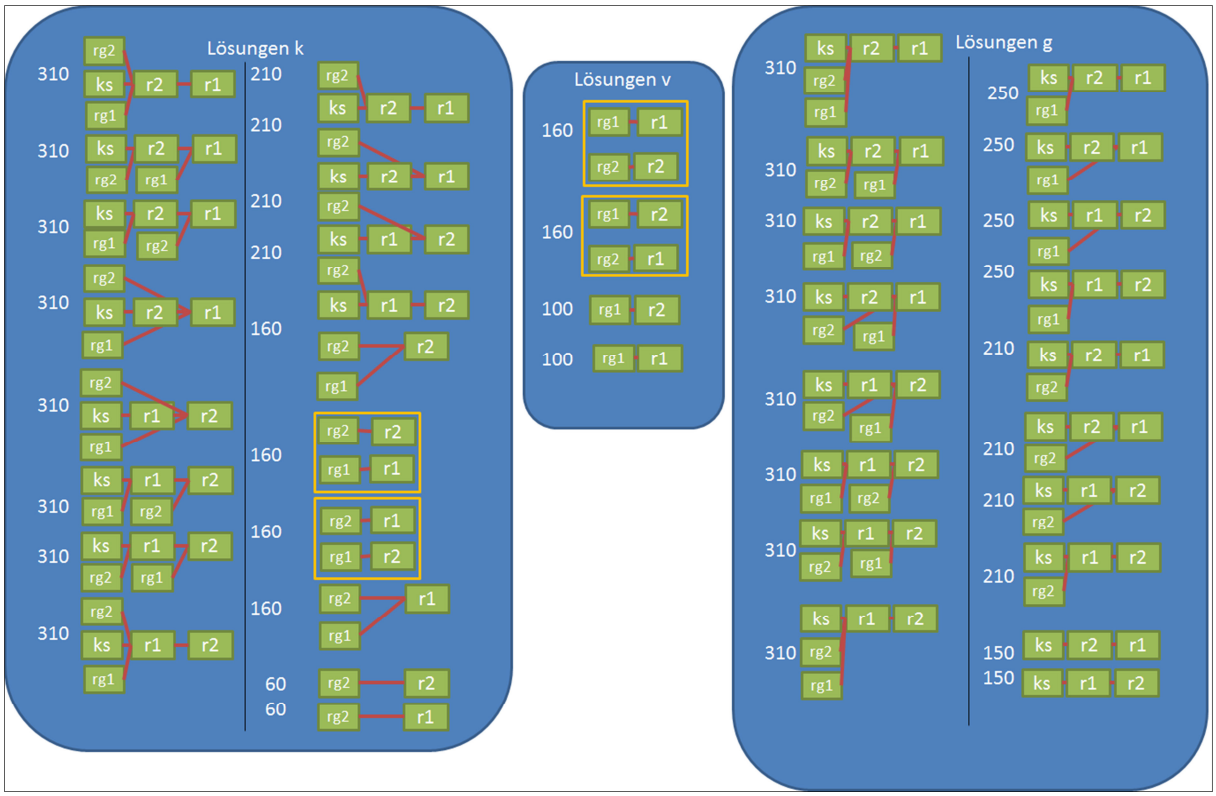

Abbildung 5-8: Listen aller möglichen lokalen Anwendungsdomänenkonfigurationen

Alle Anwendungsinstanzen wählen nun die erste Anwendungsdomänenkonfiguration aus ihrer Liste und senden sie an die an der Konfiguration beteiligten Anwendungsinstanzen mit dem zugehörigen Ressourcenwert aus der Bewertungsfunktion und dem Hashwert. Anschließend antworten die Anwendungsinstanzen entsprechend darauf. In Abbildung 5-9 werden die Veränderungen an der Liste nach dem 1. Durchlauf dargestellt.

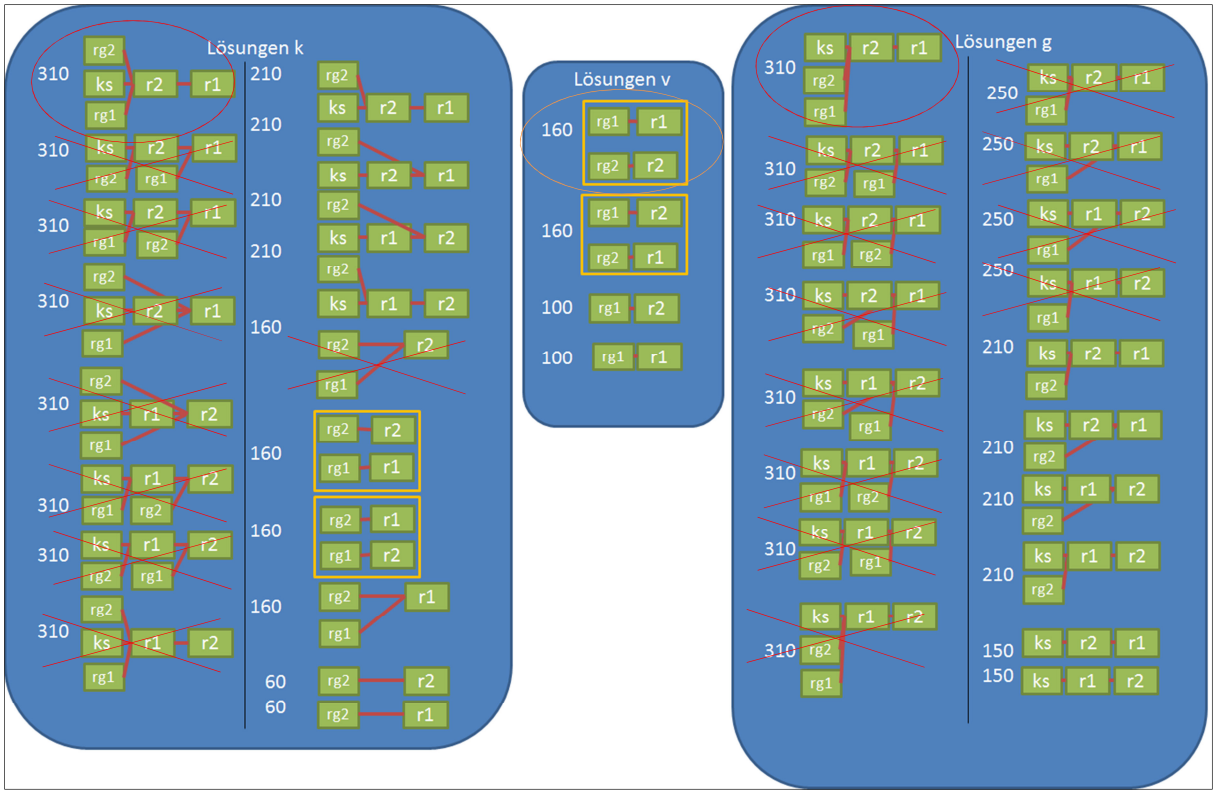

Abbildung 5-9: Reduzierte Liste aller Anwendungsdomänenkonfigurationen nach dem 1. Durchlauf

Die Ellipse illustriert, welche Anwendungsdomänenkonfiguration zum Feilschen ausgewählt wurde. Eine rote Ellipse bedeutet, dass mindestens eine beteiligte Anwendungsinstanz dieses Angebot abgelehnt hat. In diesem Beispiel hat die Anwendungsinstanz „v“ die Angebote von den anderen Anwendungsinstanzen abgelehnt. Diese Angebote besitzen einen höheren Wert als ihr eigenes Angebot und sind daher nicht in ihrer eigenen Liste enthalten. Die Anwendungsdomänenkonfigurationen von abgelehnten Angeboten werden für den nächsten Durchlauf aus den jeweiligen Listen gelöscht. Die Ellipse der Anwendungsinstanz „v“ besitzt die Farbe Orange, weil sie von der beteiligten Anwendungsinstanz „k“ noch keine Antwort erhalten hat. Die roten Kreuze bei den Anwendungsdomänenkonfigurationen der Anwendungsinstanzen „f“ und „k“ stellen die Anwendungskonfigurationen dar, welche durch das Angebot der Anwendungsinstanz „v“ gelöscht wurden. Diese Anwendungsdomänenkonfigurationen enthalten jeweils eine Anwendungskonfiguration der Anwendungsinstanz „v“, stehen aber nicht in deren Liste, was der Wert ihres Angebotes zeigt.

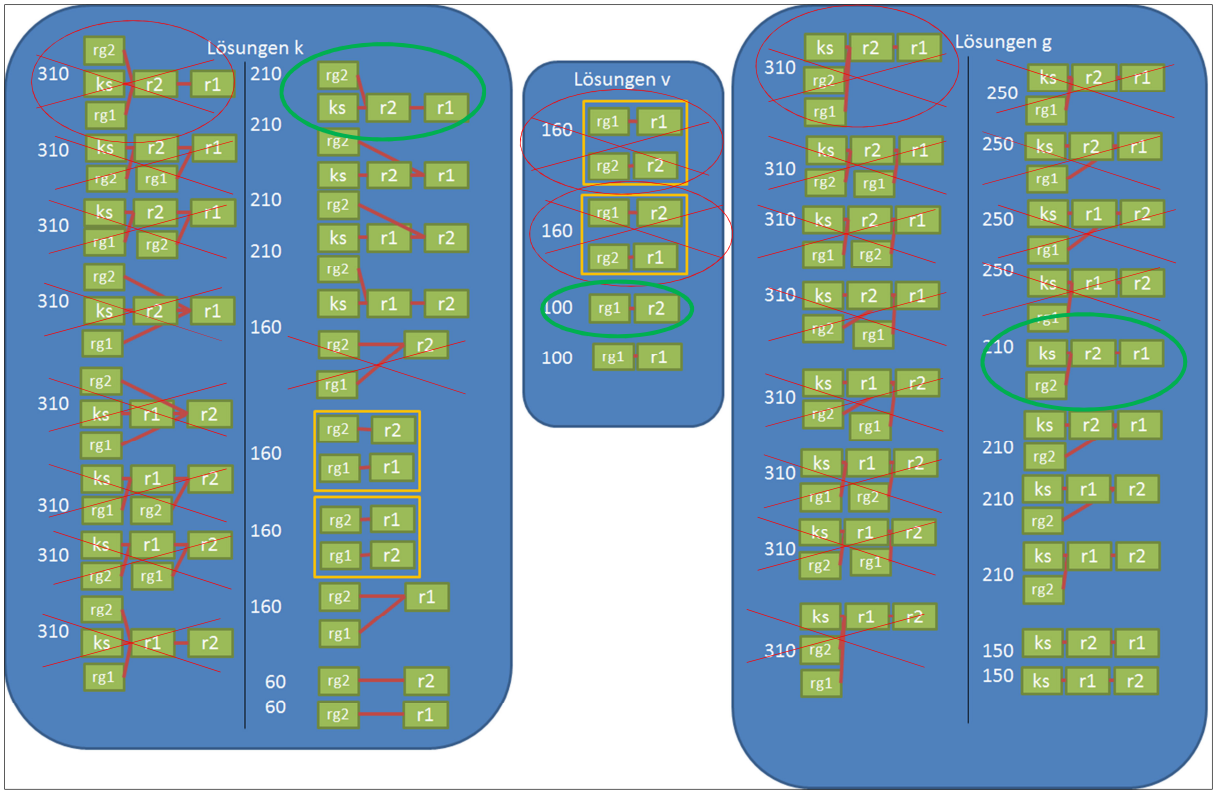

Abbildung 5-10: Liste aller Anwendungsdomänenkonfiguration mit finaler Auswahl

Im nächsten Durchlauf wählen die Anwendungsinstanzen „f“ und „k“ jeweils eine neue Anwendungsdomänenkonfiguration (siehe Abbildung 5-9). Da beide Anwendungsinstanzen die gleiche Anwendungsdomänenkonfiguration ausgewählt haben und keine weitere Anwendungsinstanz eine Anwendungskonfiguration zu der gewählten Anwendungsdomänenkonfiguration beisteuert, erhalten beide Anwendungsinstanzen gegenseitig jeweils eine Zustimmung. Beide Anwendungsinstanzen wählen die Anwendungsdomänenkonfiguration als Ergebnis dieses Zustandes und wechseln in den Zustand AUCTION. Die Anwendungsinstanz „k“ sendet nun zu allen Angeboten der Anwendungsinstanz „v“ eine Ablehnung. Daher wählt die Anwendungsinstanz „v“ schließlich die Anwendungsdomänenkonfiguration, wie in Abbildung 5-9 gezeigt, bei der nur sie selbst eine Anwendungskonfiguration beisteuert. Abschließend wechselt auch diese Anwendungsinstanz in den Zustand AUCTION.

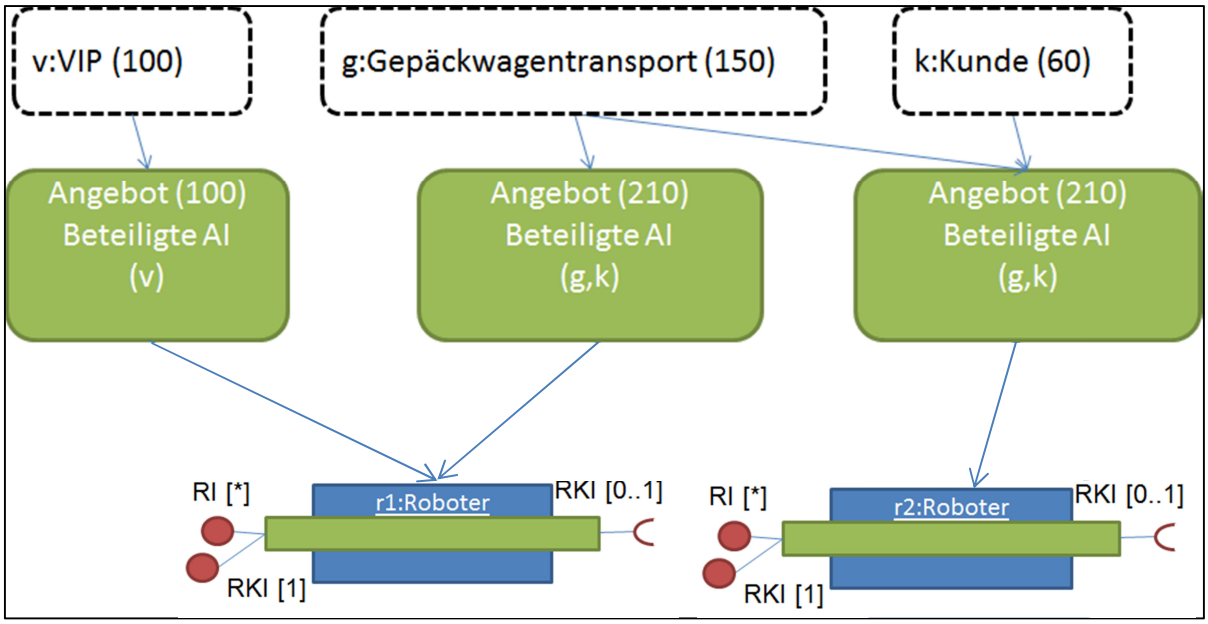

Abbildung 5-11: Angebote an die Komponenten

Im Zustand AUCTION sendet die Anwendungsinstanz „v“ an die Roboterkomponente „r2“ ihr Angebot mit dem zugehörigen Ressourcenwert. Die Anwendungsinstanzen „f“ und „k“ senden an beide Roboterkomponenten ihr Angebot aus der gemeinsamen Anwendungsdomänenkonfiguration und den dazugehörigen Ressourcenwert sowie den Hashwert. Die Roboterkomponente r1 erhält nur das Angebot der beiden Anwendungsinstanzen „f“ und „k“ und wählt dieses aus. Die Roboterkomponente „r2“ muss zwischen dem gemeinsamen Angebot der Anwendungsinstanzen „f“ und „k“ und dem Angebot der Anwendungsinstanz „v“ wählen. Die Komponente wählt das gemeinsame Angebot der Anwendungsinstanzen „f“ und „k“, weil dieses den höheren Ressourcenwert der beiden Angebote besitzt. Somit erhalten die Anwendungsinstanzen „f“ und „k“ für ihre Anwendungsdomänenkonfiguration alle benötigten Komponenten, starten die Verschaltung der Komponenten und wechseln zusammen mit diesen Komponenten in den Zustand RUNNING. Die Anwendungsinstanz „v“ erhält das Signal „cancel“ und wechselt wieder in den Zustand SEARCHING für den nächsten Versuch, eine Anwendungskonfiguration zu erstellen.

Dieses Kapitel hat das Lösungsverfahren zu der zweiten Forschungsfrage der Problemstellung aus Kapitel 3 ausführlich beschrieben und an einem Szenario verdeutlicht. Für eine Evaluation des Verfahrens wird im nächsten Kapitel ein Prototyp vorgestellt. Die Evaluation mit dem Prototyp wird anschließend im 7. Kapitel beschrieben.

6 Prototypische Implementierung der Lösung

Im vorherigen Kapitel wurde das Verfahren einer dezentralen Verschaltung von Komponenten zu Anwendungsinstanzen ausführlich beschrieben und an einem Beispiel verdeutlicht.

Dieses Kapitel beschreibt einen Prototyp als Implementierung der Lösung. Dazu wird erst der Aufbau des Prototyps erläutert und anschließend der Ablauf des Lösungsverfahrens beschrieben. Im nachfolgenden Kapitel findet dieser Prototyp Anwendung für die prototypische Evaluierung des Lösungsverfahrens.

6.1 Modell des Prototyps

Der Prototyp wurde in der Programmiersprache Java unter Berücksichtigung des Metamodells aus Abbildung 4-2 entwickelt.

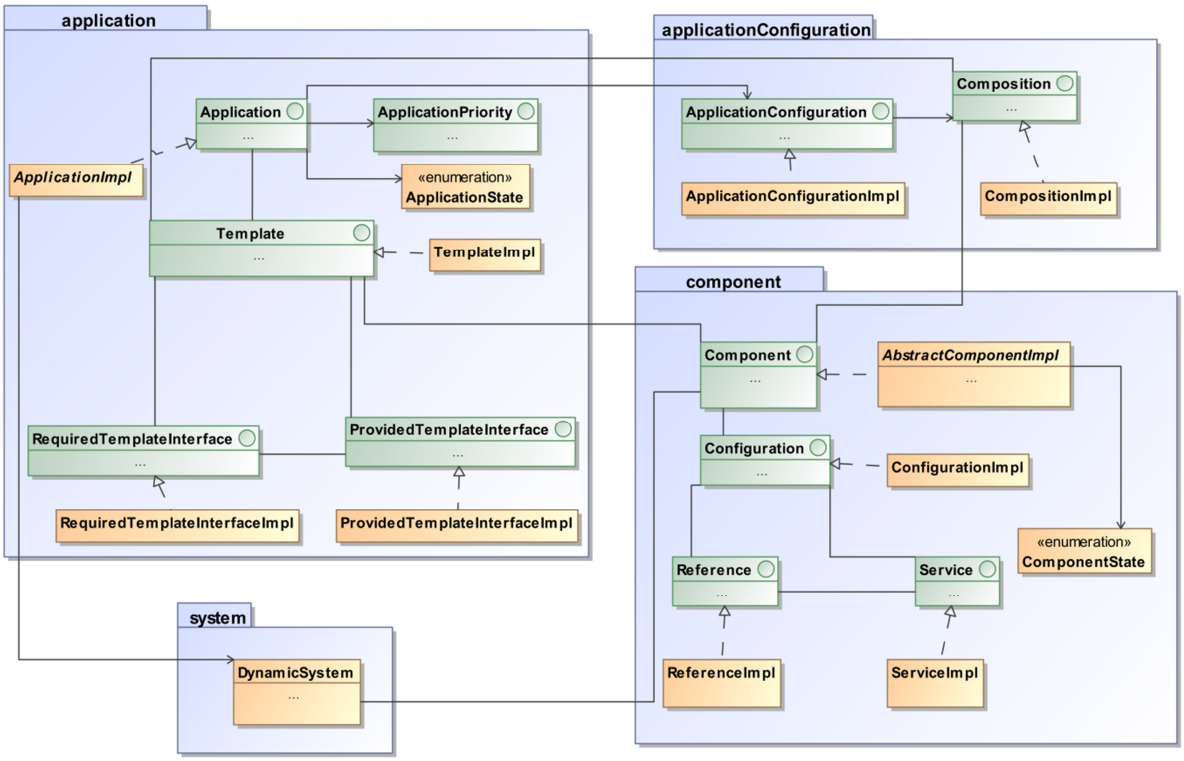

Abbildung 6-1: Modell des Prototyps

Die vier Java-Schnittstellen „Component“, „Configuration“, „Service“ und „Reference“ aus Abbildung 6-1 entsprechen dabei den Klassen „DynamicDatapiveComponent“, „ComponentConfiguration“, „ProvidedService“ und „RequiredServiceReferenceSet“ aus Abbildung 4-2. In der Java-Schnittstelle Service wird ebenfalls die Klasse „DomainInterface“ abgebildet. Java-Schnittstellen, welche als Kommunikation zwischen „DynamicDatapiveComponents“ verwendet werden, müssen demnach von der Java-Schnittstelle Service abgeleitet sein.

Eine Anwendung wird über die vier Java-Schnittstellen „Application“, „Template“, „ProvidedTemplateInterface“ und „RequiredTemplateInterface“ umgesetzt, welche in ihrer Struktur der Anwendungsbeschreibung einer Anwendung aus dem Metamodell gleichen. Wie auch im Metamodell besitzt eine Anwendung des Prototyps ebenfalls eine Priorität, welche über die Java-

Schnittstelle „ApplicationPriority“ abgebildet wird. Auch die Zustände von Anwendungsinstanzen und Komponenten aus Kapitel 5 sind in dem UML-Klassendiagramm enthalten.

Eine Instanz der Schnittstelle „ApplicationConfiguration“ in Abbildung 6-1 entspricht der Definition einer Anwendungskonfiguration, wenn die Instanz mit nur einer Anwendung verknüpft ist. Bei einer Verknüpfung von mehreren Anwendungen entspricht die Instanz der Definition einer Anwendungsdomänenkonfiguration und bei Verknüpfung aller gestarteten Anwendungen im System der Definition einer Systemkonfiguration.

Die Klasse „DynamicSystem“ entspricht der Definition des Systems aus Abbildung 4-2. Sie ermöglicht es, dass Anwendungsinstanzen in Schritt 1 des Verfahrens an alle existierenden Komponenten des Systems eine Anfrage senden können. In den nachfolgenden Abschnitten werden die Methoden der einzelnen Klassen, welche für das Lösungsverfahren aus Kapitel 5 benötigt werden, kurz beschrieben.

6.1.1 Teilmodell System

Das Teilmodell „System“ aus Abbildung 6-2 beinhaltet nur die Klasse „DynamicSystem“. Sie ist für die Weiterleitung von Anfragen von Anwendungsinstanzen aus Schritt 1 des Verfahrens an alle im System existierenden Komponenten zuständig.

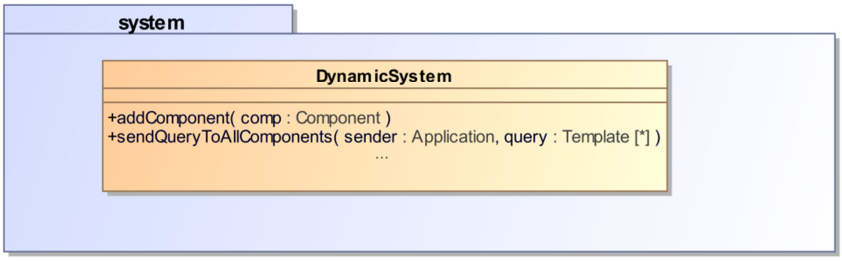

Abbildung 6-2: Teilmodell System des Prototyps

Mit der Methode „addComponent“ können sich Komponenten bei dieser Klasse registrieren. Über die Methode „sendQueryToAllComponents“ werden Nachrichten an die registrierten Komponenten anschließend weitergeleitet.

6.1.2 Teilmodell ApplicationConfiguration

Das Teilmodell in Abbildung 6-3 enthält zwei Schnittstellen mit jeweils einer Klasse pro Schnittstelle. Jede „Composition“ enthält eine Referenz auf genau eine Komponente, eine Menge von „Templates“ und eine Menge von Referenzen zu anderen Instanzen vom Typ „Composition“. Eine Instanz dieser Klasse stellt eine Zuordnung einer Komponente zu dieser Menge von „Templates“ dar. Außerdem enthält eine Instanz dieser Klasse alle Verschaltungen zu anderen Komponenten mit der jeweiligen Schnittstelle (DomainInterface). In jeder „ApplicationConfiguration“ existiert für jede verwendete Komponente genau eine Instanz vom Typ „Composition“. Über die Methode „getReferencedComponent“ kann die Komponente einer „Composition“ zurückgegeben werden. Über „getInterfaceNames“ werden die Namen der „DomainInterfaces“ der Komponente zurückgegeben, über die anschließend eine Menge von zugehörigen „Compositions“ ausgelesen werden kann. Bei Umsetzung der Konfiguration wird die Komponente den „Templates“ zugeordnet und anschließend mit den Komponenten der eben beschriebenen „Compositions“ über die „DomainInterfaces“ verschaltet.

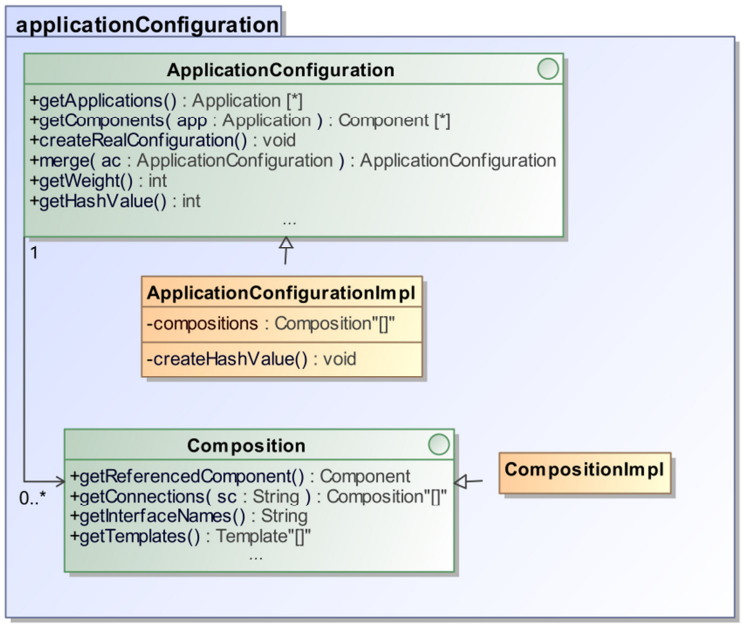

Abbildung 6-3: Teilmodell ApplicationsConfiguration des Prototyps

Eine Instanz der Klasse „ApplicationConfiguration“ enthält eine Menge von Instanzen der Klasse „Composition“ und stellt eine Anwendungskonfiguration oder auch eine Anwendungsdomänenkonfiguration dar. Über die „Get“-Methoden dieser Klasse können die Menge der zugehörigen Anwendungsinstanzen, die Menge an Komponenten, die Gewichtung und der Hash-Wert der Anwendungskonfiguration ausgelesen werden. Mit der Methode „createRealConfiguration“ wird die Anwendungskonfiguration, wie im letzten Absatz beschrieben, umgesetzt. Vorher werden allerdings noch die bis dahin bestehenden Zuordnungen der Komponenten gelöst. Diese Komponenten wechseln dabei von einer bestehenden Anwendungskonfiguration in diese Anwendungskonfiguration.

Mit der Methode „merge“ werden zwei Mengen vom Typ „Composition“ zu einer Menge zusammengefügt. Sollten dabei zwei „Compositions“ die gleiche Komponente referenzieren, werden sie zu einer zusammengefasst. Das bedeutet, dass die daraus resultierende „Composition“ alle „Templates“ und Verbindungen beider „Compositions“ enthält.

6.1.3 Teilmodell Application

Das Teilmodell „Application“ in Abbildung 6-4 beinhaltet alle Schnittstellen und Klassen, welche zusammen eine Anwendung darstellen. Eine Instanz vom Typ „Application“ entspricht dabei einer Anwendungsinstanz. Die „Get“-Methoden liefern hierbei Informationen und Referenzen auf Instanzen zurück. Zum Beispiel liefert die Methode „getMinNoOfRequiredComponents“ die Anzahl an Komponenten als Wert zurück, wie viele Komponenten mindestens dem „Template“ zugeordnet werden müssen, damit die Anwendung laufen kann. Die restlichen Methoden werden bei der nachfolgenden Beschreibung des implementierten Verfahrens beschrieben.

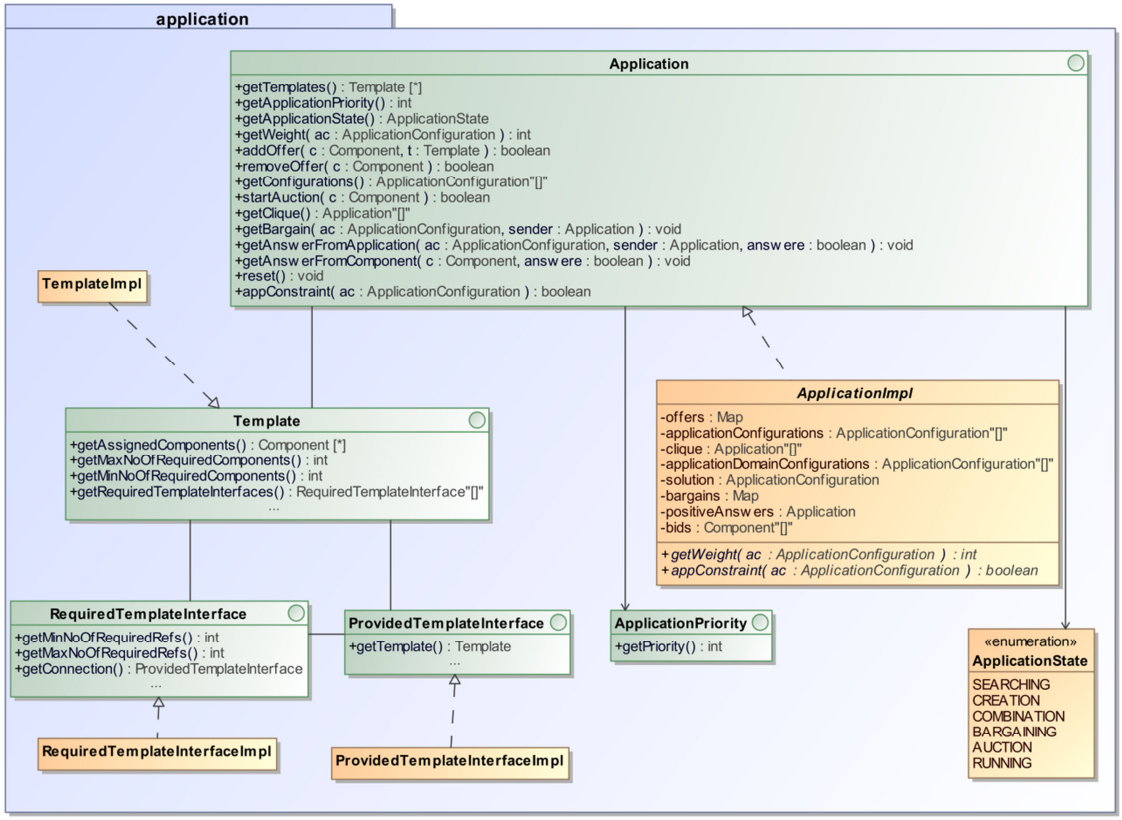

Abbildung 6-4: Teilmodell Application des Prototyps

Eine Instanz vom Typ „Application“ besitzt außerdem noch eine Menge von Attributen, welche im Verfahren verwendet werden. Das Attribut „offers“ vom Typ „Map“ enthält alle in Schritt 1 des Verfahrens erhaltenen Angebote der Komponenten. Eine „Map“ arbeitet nach dem „Key-Value“-Verfahren, wobei über die Angabe des „Key“ der „Value“ gelesen oder bearbeitet werden kann. Im Fall des Attributs „offers“ ist der „Key“ vom Typ „Template“ und „Value“ beinhaltet eine Liste von Komponenten, welche diesem „Template“ aus „Key“ zugeordnet werden können. Die Attribute „applicationConfigurations“ und „applicationDomainConfigurations“ enthalten jeweils eine Menge von Anwendungskonfigurationen bzw. Anwendungsdomänenkonfigurationen. Das Attribut „clique“ beinhaltet die Menge von Konkurrenten vom Typ „Application“. Im Attribut „solution“ wird die Anwendungsdomänenkonfiguration nach dem 4. Schritt des Lösungsverfahrens gespeichert. Die Attribute „bargains“ vom Typ „Map“ und „positiveAnswers“ werden ebenfalls im 4. Schritt verwendet. Das Attribut „bargains“ enthält noch unbeantwortete aktuelle Angebote von anderen Anwendungsinstanzen und das Attribut „positiveAnswers“ die Zustimmungen anderer Anwendungsinstanzen zu dem aktuellen Angebot dieser Anwendungsinstanz. Das Attribut „bids“ wird im 5. Schritt des Lösungsverfahrens verwendet und enthält eine Liste von Komponenten, welche das Gebot dieser Anwendungsinstanz gewählt haben.

6.1.4 Teilmodell Component

Das Teilmodell „Component“ (Abbildung 6-5) enthält alle Klassen und Schnittstellen, welche zusammen eine Komponente darstellen. Über die Methode „connect“ kann eine Komponente einem „Template“ zugeordnet oder mit einer anderen Komponente verschaltet werden. Mit der Methode „disconnectAll“ werden diese Verbindungen wieder gelöst. Die übrigen Methoden werden wie auch beim Teilmodell „Application“ bei der nachfolgenden Beschreibung des implementierten Verfahrens beschrieben.

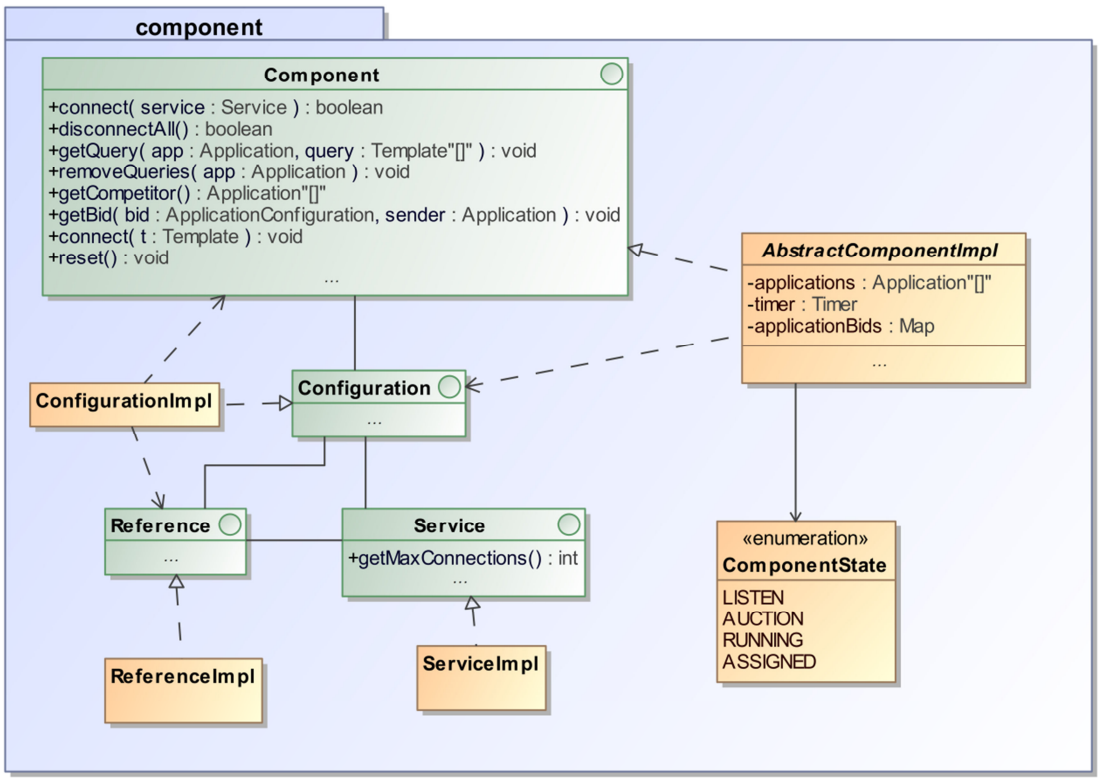

Abbildung 6-5: Teilmodell Component des Prototyps

Eine Instanz vom Typ „Component“ besitzt zusätzlich noch drei Attribute, welche beim Lösungsverfahren verwendet werden. Das Attribut „applications“ enthält eine Menge von Anwendungsinstanzen, welche im 1. Schritt des Lösungsverfahrens von dieser Komponente ein Angebot erhalten haben. Mit dem Attribut „timer“ wird ein Timer verwendet. Über den Timer wird der Zeitpunkt bestimmt, wann die Komponente die Methode „startAuction“ einer Auswahl von Anwendungsinstanzen aufrufen soll. Das Attribut „applicationBids“ vom Typ „Map“ enthält alle Gebote der Anwendungsinstanzen, welche im 5. Schritt des Verfahrens gesendet wurden.

6.2 Das implementierte Verfahren

Im letzten Abschnitt wurden die Struktur des Prototyps und eine Auswahl der Methoden der enthaltenen Klassen und Schnittstellen sowie die im Verfahren verwendeten Attribute beschrieben. In diesem Abschnitt wird das implementierte Verfahren erläutert.

6.2.1 Die Zustände

Jeder Akteur vom Typ „Application“ oder „Component“ besitzt einen eigenen Ablauf von Aktivitäten, welche der Akteur selbstständig ausführt. Für beide Typen wird nachfolgend der jeweilige Ablauf über ein Aktivitätsdiagramm dargestellt und kurz beschrieben. Die Aktivitäten werden in die Zustände der Akteure eingeordnet und entsprechen dem beschriebenen Verfahren aus Kapitel 5.

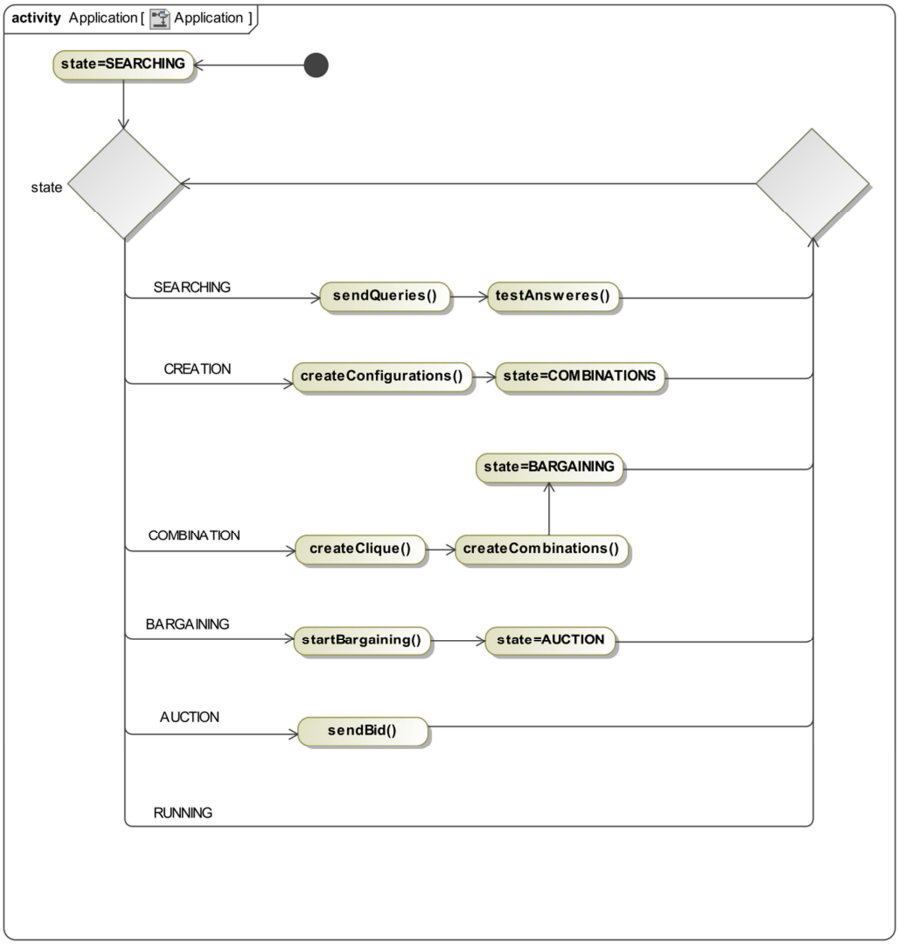

Abbildung 6-6: Ablauf einer Anwendung im Prototyp

In Abbildung 6-6 wird der Ablauf der Aktivitäten einer Instanz vom Typ „Application“ dargestellt. Die Aktivitäten werden je nach aktuellem Zustand der Instanz ausgewählt. Zum Beispiel wird im Zustand SEARCHING die Methode „sendQueries“, gefolgt von der Methode „testAnswers“ aufgerufen. Anschließend wird der aktuelle Zustand für die nächsten Aktivitäten geprüft. Zu Beginn des Ablaufs befindet sich jede Instanz vom Typ „Application“ im Zustand SEARCHING.

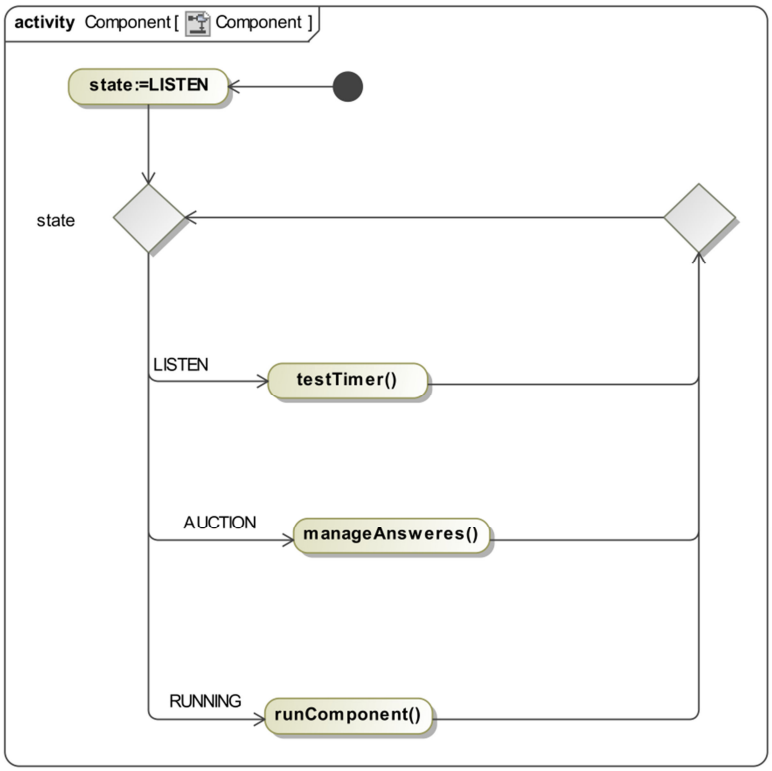

Abbildung 6-7: Ablauf einer Komponente im Prototyp

In Abbildung 6-7 wird der Ablauf der Aktivitäten einer Instanz vom Typ „Component“ dargestellt. Jede Instanz vom Typ „Component“ beginnt im Zustand LISTEN. Analog zu dem Ablauf der Aktivitäten von Instanzen vom Typ „Application“ wird ebenfalls für jeden Zustand eine Auswahl an Methoden ausgeführt, bevor der Zustand erneut geprüft wird. Anders als beim Ablauf einer Anwendungsinstanz beinhaltet der Zustand RUNNING hier aber die Methode „runComponent“. In dieser Methode steht der komponentenabhängige Programmcode, welcher nach Verschaltung der Komponente zyklisch ausgeführt wird.

6.2.2 Implementierung des Verfahrens im Überblick

In Abbildung 6-8 wird der Ablauf des Verfahrens an dem 1. Szenario des Anwendungsbeispiels analog zu Abbildung 5-3 dargestellt. Allerdings werden hier die Methoden präsentiert, welche die Aufrufe aus Abbildung 5-3 realisieren.

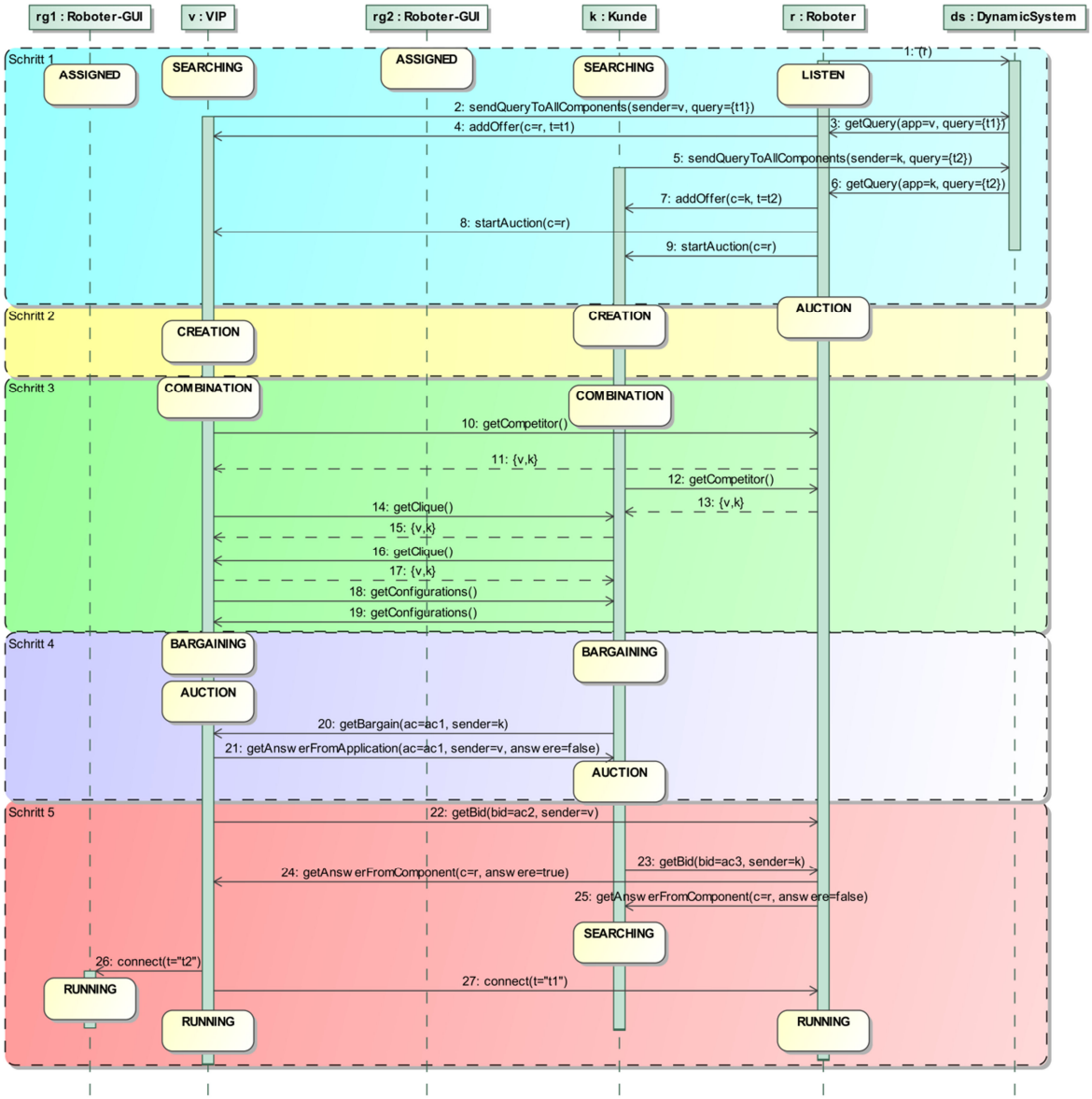

Abbildung 6-8: Kommunikation zwischen den implementierten Akteuren am Beispiel von Szenario 1 des Anwendungsbeispiels

Nachfolgend wird jeder Schritt des implementierten Lösungsverfahrens einzeln erläutert. Dabei werden ebenfalls die verwendeten Methoden beschrieben und bei komplexeren Methoden zusätzlich in Pseudocode dargestellt.

6.2.3 Implementierung Schritt 1

Im ersten Schritt des Verfahrens sollen, wie in Kapitel 5 beschrieben, die nötigen Informationen zwischen Anwendungsinstanzen und Komponenten ausgetauscht werden. Die Anwendungsinstanzen befinden sich dabei im Zustand SEARCHING und die freien Komponenten im Zustand LISTEN, siehe Abbildung 6-8. In der Methode „sendQueries“ ermittelt jede Anwendungsinstanz aus ihren „Templates“ diejenigen, welche noch Komponenten benötigen. Dafür vergleicht sie bei jedem „Template“, ob dessen zugeordnete Komponentenzahl und die Zahl der Komponenten, die bis dahin positiv geantwortet haben, mindestens der minimal geforderten Zahl des „Templates“ entsprechen. Bei den „Templates“, für welche dies nicht der Fall ist, wird „sendQueryToAllComponents“ der Klasse „DynamicSystem“ aufgerufen, siehe Abbildung 6-9. Die Variable „offers“

enthält dabei alle bis zu diesem Zeitpunkt eingegangenen Angebote von Komponenten zu den einzelnen „Templates“.

```
offers; //enthält alle bisherigen Angebote von Komponenten nach Templates sortiert.

sendQueries()

        temp; //leere Liste

        forall t aus getTemplates() do

                if (t.getMinNoOfRequiredComponents().size() >t.getAssignedComponents().size()

                  + offers(t).size() ) do

                        temp.add(t)

                end

        end

        if (temp.size() >0) do

                DynamicSystems.sendQueryToAllComponents(this,temp)

        end
```

Abbildung 6-9: Pseudocode zu sendQueries()

Anschließend wird in „testAnswers“ geprüft, ob alle Komponenten aus „offers“ die Methode „startAuction“ aufgerufen haben. Wenn dies der Fall ist, wird analog zu „sendQueries“ geprüft, ob zu allen „Templates“ genug Komponenten vorhanden sind. Ist dies der Fall, wechselt die Anwendungsinstanz in den Zustand CREATION. Wenn zu diesem Zeitpunkt hingegen nicht genug Komponenten vorhanden sind, wird bei allen Komponenten in „offers“ die Methode „removeQueries“ aufgerufen. Da anscheinend nicht genug Komponenten für eine Anwendungskonfiguration vorhanden sind, nimmt diese Anwendungsinstanz bei diesem gestarteten Verschaltungsprozess nicht Teil. Anschließend wird „offers“ geleert. Danach oder wenn noch nicht alle Komponenten aus „offers“ die Methode „startAuction“ aufgerufen haben, wird im nächsten Durchlauf im Zustand SEARCHIN wieder die Methode „sendQueries“ aufgerufen.

Bei Komponenten, welche sich über „addComponent“ bei der Klasse „DynamicSystem“ registriert haben, wird durch die Methode „sendQueryToAllComponents“ ihre Methode „getQuery“ aufgerufen. Wenn sich die Komponente im Zustand LISTEN befindet, prüft sie, in welchen „Templates“ sie verwendet werden kann, siehe Definition aus Kapitel 3. Die Prüfung wird ebenfalls ausgeführt, wenn sich die Komponente im Zustand RUNNING befindet und die anfragende Anwendungsinstanz eine höhere Priorität besitzt als die aktuell mit ihr verbundenen Anwendungsinstanzen. Zusätzlich wechselt die Komponente in den Zustand LISTEN. Wenn die Komponente bei mindestens einem „Template“ verwendet werden kann, werden drei Aktionen ausgeführt. Zuerst wird für jedes positiv getestete „Template“ der anfragenden Anwendung dies über die Methode „addOffer“ mitgeteilt. Die Anwendungsinstanz speichert die Antwort in ihre Variable „offers“. Anschließend speichert die Komponente die Anwendungsinstanz in der Variablen „applications“ und startet einen Timer. Wenn die Komponente schon einen Timer gestartet hat, wird dieser wieder auf null gesetzt.

```
applications;   // enthält alle Anwendungsinstanzen, wo mind. Ein Template genutzt werden kann.

testTimer()

        if {Timer ist gestartet und abgelaufen} do

                {stoppe Timer}

                do

                        temp := {Teilmenge aus applications mit der höchsten Priorität}

                        forall app from temp do

                                if ( app.startAuction() == false)

                                        temp.remove(app)

                                        applications.remove(app)

                                end

                        end

                while(temp.size()==0 && applications.size()>0)

                if(temp.size()>0)

                        forall app from application do

                                if ( app is not in temp) do

                                        app.removeOffer(this)

                                end

                        end

                        application:=temp

                        state :=CREATION

                end

        end
```

Abbildung 6-10: Pseudocode zu testTimer()

Wie Anwendungsinstanzen besitzen auch Komponenten einen eigenen Prozess, welcher pro Zustand eine Menge von Methoden enthalten kann. In dem Zustand LISTEN ist dies die Methode „testTimer“, siehe Abbildung 6-10. Diese Methode prüft, ob der Timer der Komponente gestartet und abgelaufen ist. Wenn der Timer abgelaufen ist, wird er gestoppt und die Komponente ermittelt die Anwendungsinstanzen mit der höchsten Priorität aus der Variablen „applications“. Bei diesen ruft sie anschließend die Methode „startAuction“ auf. Die Anwendungsinstanzen, welche den Aufruf ablehnen, werden aus „applications“ gelöscht. Eine Ablehnung bedeutet in diesem Fall, dass die Anwendungsinstanz nicht mehr existiert. Wenn alle Anwendungsinstanzen ablehnen, werden die Anwendungsinstanzen mit der nächstniedrigeren Priorität gesucht und deren „startAuction“-Methode wird aufgerufen. Wenn die Variable „applications“ dabei irgendwann keine Anwendungsinstanzen mehr besitzt, wird die Methode beendet. Bei Zusage einer Anwendungsinstanz hingegen werden alle nicht ausgewählten Anwendungsinstanzen aus „applications“ gelöscht und der Zustand der Komponente wird in AUCTION geändert.

Abschließend muss noch geklärt werden, was bei dem Aufruf der Methode „removeQueries“ der Komponente geschieht. In dieser Methode wird die übergebene Anwendungsinstanz aus der Variablen „applications“ gelöscht. Sollte dies die letzte Anwendungsinstanz in der Variablen gewesen sein und die Komponente sich schon im Zustand AUCTION befinden, wechselt sie wieder in den Zustand LISTEN.

6.2.4 Implementierung Schritt 2

In diesem Schritt erstellt eine Anwendungsinstanz im Zustand CREATION alle möglichen Anwendungskonfigurationen. Dafür wird die Methode „createConfigurations“ ausgeführt, nach deren Ende die Anwendungsinstanz sofort in den Zustand COMBINATION wechselt.

Die Methode „createConfigurations“ berechnet alle möglichen Anwendungskonfigurationen in zwei Schritten. Im ersten Schritt wird für jedes „Template“ die Menge der möglichen Zuordnungen von Komponenten ermittelt. Anschließend wird für jedes „Template“ eine mögliche Zuordnung ausgewählt und im zweiten Schritt zu dieser Auswahl werden alle möglichen Verschaltungen zwischen den Komponenten erstellt. Zu jeder Verschaltung wird anschließend ein Objekt vom Typ „ApplikationConfiguration“ erzeugt.

```
createConfigurations()

        powerSets; // Menge von Potenzmengen für die Templates.

        forall  t aus getTemplates() do

                min := t.getMinNoOfRequiredComponents()-t.getAssignedComponents().size()

                max := t.getMaxNoOfRequiredComponents()-t.getAssignedComponents().size()

                max := MIN(max, offers(t).size())

                if (t.getMaxNoOfRequiredComponents() == 0) do

                        max := offers(t).size()

                end

                temp := ∀ p ⊆ offers(t) : min<=|p|<=max

                forall p aus temp do

                        p:=p+t.getAssignedComponents()

                end

                powerSets(t):= temp

        end

        chosen := {Menge aller Tupel der Länge powerSets.size(). Jedes Element eines Tupels enthält
        ein Element eines anderen Templates aus powerSets. Es existieren keine Duplikate von Tupel
        in dieser Menge.}

        forall k from chosen do

                test := {prüfe, ob jede beteiligte Komponente in der Auswahl genau einem Template
                dieser Anwendungsinstanz zugeordnet ist.}

                If (test == wahr) do

                        createConnections(k)

                end

        end
```

Abbildung 6-11: Pseudocode zu createConfigurations()

In Abbildung 6-11 wird die Methode „createConfigurations“ abgebildet. Zuerst wird für jedes „Template“ der Anwendungsinstanz die Menge der noch benötigten Komponenten ermittelt. Anschließend werden aus der Menge der Angebote „offers“ Teilmengen ermittelt, welche die benötigte Anzahl von Komponenten enthalten. Zu jeder Teilmenge werden dann die diesen „Templates“ fest zugeordneten Komponenten hinzugefügt. Damit enthält jede Teilmenge einen vollständigen Satz an Komponenten, welche dem „Template“ zugeordnet werden können. Die Menge an Teilmengen ist selbst auch vollständig, d. h. jede mögliche Teilmenge von Komponenten, welche diesem „Template“ zugeordnet werden kann, befindet sich in dieser Menge.

Anschließend wird für jedes „Template“ aus dessen Menge eine Teilmenge ausgewählt und zu einem Tupel zusammengefügt. Für jede Kombination des Tupels wird geprüft, ob jede enthaltene Komponente, wie gefordert, genau einem „Template“ dieser Anwendungsinstanz zugeordnet ist. Wenn dies der Fall ist, wird das Tupel an die Methode „createConnections“ übergeben.

```
applicationConfigurations; //Menge aller möglichen Anwendungskonfigurationen

createConnections(chosen)

        templates; //Menge aller möglichen Verbindungen pro Template

        forall template from chosen do

                requires; //Menge von Potenzmengen für RequiredTemplateInterfaces

                forall req from template.getRequiredTemplateInterfaces() do

                        if (req.getConnection()<>null) do

                                t := req.getConnection().getTemplate()

                                min := req.getMinNoOfRequiredRefs()

                                max := req.getMaxNoOfRequiredRefs()

                                if (req.getMaxNoOfRequiredRefs()==0) do

                                        max := chosen(t).size()

                                end

                                temp := ∀ p ⊆ chosen(t) : min<=|p|<=max

                                requires(req):=temp

                        end

                end

                templates(template) := {Menge aller Tupel der Länge requires.size(). Jedes Element
                eines Tupels enthält ein Element eines anderen RequiredTemplateInterfaces aus requi-
                res. Es existieren keine Duplikate von Tupel in dieser Menge.}

        end

        connections := {Menge aller Tupel der Länge aller Komponenten in chosen. Jedes Element
        eines Tupels enthält ein Element des zugehörigen Template aus templates. Es existieren kei-
        ne Duplikate von Tupel in dieser Menge.}

        forall c from connections do

                test:= Jede Komponente wurde nur so oft verschaltet, wie getMaxConnections() ihres
                Dienstes angibt AND jeder angebotene Dienst einer Komponente wurde mindestens einmal
                verschaltet, wenn bei dem zugehörigen Template die entsprechende ProvidedTemplateIn-
                terface mit einem anderen Template verbunden ist.

                if (test == wahr) do

                        adc:= new ApplicationConfigurationImpl(c,chosen)

                        if ( appConstraint(adc) == true) do

                                applicationConfigurations.add(adc)

                        end

                end

        end
```

Abbildung 6-12: Pseudocode zu createConnections()

Die Methode in Abbildung 6-12 berechnet als zweiten Schritt von „createConfigurations“ alle möglichen Verschaltungen des übergebenen Tupels. Dafür wird für jedes „RequiredTemplateInterface“ jedes „Template“ im übergebenen Tupel darauf geprüft, ob es mit einem anderen „Template“ verbunden ist. Anschließend werden Minimalanzahl und Maximalanzahl an benötigten Verbindungen ermittelt. Analog zu „createConfigurations“ wird nun jede mögliche Teilmenge aus Komponenten erstellt, welche die Anzahl der benötigten Verbindungen erfüllen. Dann wird aus jedem „RequiredTemplateInterface“ eines „Templates“ ein Element dieser Verbindungen gewählt und in einem weiteren Tupel zusammengefasst. Die Menge aller möglichen Tupel wird anschließend in der Variable templates dem betrachteten „Template“ zugeordnet. Nun wird für jede Komponente aus dem übergebenen Tupel das zugehörige „Template“ ermittelt und ein entsprechendes Element aus „templates“ genommen. Diese Elemente werden in einem dritten Tupel zusammengefasst. Dieses Tupel entspricht einer möglichen Verschaltung dieser Komponenten. Nun wird für jede Kombination des Tupels geprüft, ob diese Verschaltungen auch alle Kriterien erfüllen. Jeder Dienst jeder Komponente darf nur so oft verschaltet werden, wie dieser Dienst auch angeboten werden kann. Außerdem muss jede Komponente auch mindestens einmal verschaltet worden sein, wenn das zugehörige „Template“ einem anderen „Template“ einen Dienst anbietet. Ansonsten würde die Komponente in dieser Verschaltung nicht benötigt und könnte dementsprechend aus dieser Anwendungskonfiguration gelöscht werden. Falls diese Kombination gültig ist, würde sie im ersten Schritt allerdings erstellt werden und diese Verschaltung wäre demnach ein Duplikat. Mit diesem Kriterium kann solch ein Duplikat vermieden werden.

Wenn die Kriterien bei einer Kombination erfüllt sind, wird nun eine Instanz vom Typ „ApplicationConfiguration“ mit dieser Kombination erstellt. Erfüllt die Instanz auch die Bedingungen der Methode „appConstraint“, wird sie in der Liste der Anwendungskonfigurationen dieser Anwendungsinstanz gespeichert. Damit wurden alle möglichen Anwendungskonfigurationen dieser Anwendungsinstanz ermittelt.

6.2.5 Implementierung Schritt 3

Im dritten Schritt befinden sich die Anwendungsinstanzen im Zustand COMBINATION. In diesem Zustand werden zwei Aktionen ausgeführt. Zuerst wird eine Liste aller potenziellen Konkurrenten, die sogenannte Clique, aufgestellt. Anschließend wird gewartet, bis sich jeder Konkurrent ebenfalls im Zustand COMBINATION befindet. Nachdem dies der Fall ist, holt sich die Anwendungsinstanz alle Anwendungskonfigurationen aller Konkurrenten und bildet damit jede mögliche Anwendungsdomänenkonfiguration. Dabei behält sie nur diejenigen, bei denen ihre Bedingungen erfüllt sind. Abschließend wird diese Menge noch nach der Gewichtung und dem Hash-Wert der Konfiguration sortiert und die Anwendungsinstanz wechselt danach in den Zustand BARGAINING.

```
clique; //Liste aller in der Clique vorhandenen Konkurrenten. Ist zu Beginn leer.

createClique()

        tempClique; //Leere Listen von Anwendungsinstanzen

        forall template from offers do

                forall component from offers(template) do

                        temp := component.getCompetitor()

                        forall app from temp do

                                if (app not in tempClique) do

                                        tempClique.add(app)

                                end

                        end

                end

        end

        clique.addAll(tempClique)

        while (tempClique.size() > 0) do

                application := tempClique.getFirst()

                if (application.getClique().size()>0) do

                        forall app from application.getClique() do

                                if (app not in clique) do

                                        clique.add(app)

                                        tempClique.add(app)

                        end

                        tempClique.removeFirst()

                end

        end
```

Abbildung 6-13: Pseudocode zu createClique()

In Abbildung 6-13 wird der Pseudocode zur Erstellung der Clique von Konkurrenten dargestellt. Zuerst werden alle direkten Konkurrenten über die Komponenten ermittelt, welche Angebote geliefert haben. Da nicht jede Anwendungsinstanz die gleichen Komponenten benötigt, haben sie auch nicht immer die gleichen Konkurrenten in ihrer Clique. Da aber jeder Teilnehmer einer Clique jeden Teilnehmer direkt kennen muss, werden im zweiten Teil der Cliquenberechnung von jedem direkten Konkurrenten dessen Konkurrenten geholt. Dies wird ebenfalls bei neu gefundenen Konkurrenten ausgeführt, bis kein neuer Konkurrent gefunden wurde. Wenn ein Konkurrent selbst noch keine Konkurrenten besitzt, scheint er noch nicht in diesem Zustand angekommen zu sein. Daher wartet die Cliquenberechnung, bis der Konkurrent den Zustand erreicht hat.

```
applicationDomainConfigurations;

createCombinations()

        applicationDomainConfigurations.addAll(applicationConfigurations)

        forall application from clique do

                if (application not this) do

                        temp; //Menge an Anwendungsdomänenkonfigurationen

                        forall ac from application.getConfigurations() do

                                forall ac2 from applicationDomainConfigurations do

                                        acNew := ac2.merge(ac)

                                        if(appConstraint(acNew)) do

                                                temp.add(acNew)

                                        end

                                end

                        end

                        applicationDomainConfigurations.addAll(temp)

                end

        end
```

Abbildung 6-14: Pseudocode zu createCombinations()

In Abbildung 6-14 wird der Pseudocode zur Berechnung der Anwendungsdomänenkonfigurationen dargestellt. Zuerst wird jede Anwendungskonfiguration dieser Anwendungsinstanz als Anwendungsdomänenkonfiguration gespeichert. Anschließend wird jede gespeicherte Anwendungsdomänenkonfiguration mit jeder Anwendungskonfiguration eines Konkurrenten kombiniert. Wenn die Bedingungen dieser Anwendungsinstanz bei dem Kombinationsergebnis erfüllt sind, wird diese Kombination zwischengespeichert. Nachdem alle Kombinationen einer Anwendungsinstanz berechnet wurden, werden die zwischengespeicherten Kombinationen den gespeicherten Anwendungsdomänenkonfigurationen hinzugefügt. Anschließend wird der Vorgang bei dem nächsten Konkurrenten mit der erweiterten Menge an Anwendungsdomänenkonfigurationen wiederholt, bis alle Konkurrenten bearbeitet wurden.

Die abschließende Sortierung der Variable „applicationDomainConfigurations“ erfolgt nach folgendem Prinzip. Die Gewichtung jedes Elements der Liste ist größer als oder gleich groß wie das nächste Element der Liste. Wenn die Gewichtungen beider benachbarten Elemente gleich groß sind, ist der Hash-Wert des betrachteten Elements größer als der Hash-Wert des nächsten Elements der Liste. Die Berechnung des Hash-Wertes wird in Abbildung 6-15 dargestellt.

```
hashValue;  //Der HashWert

createHashValue()

        sortedList := sort(compositions)

        string;  //Eine Variable vom Typ String

        forall composition from sortedList do

                string:=string+composition.getReferencedComponent().hashCode()

                forall interfaceName from composition.getInterfaceNames() do

                        string:=string+interfaceName

                        sortedList2 := sort(composition.getConnectionsAsReference(interfaceName))

                        forall comp from sortedList2 do

                                string:=string+comp.getReferencedComponent().hashCode()

                        end

                end

        end

        hashValue := string.hashCode()
```

Abbildung 6-15: Pseudocode zu createHashValue()

6.2.6 Implementierung Schritt 4

In diesem Schritt befindet sich die Anwendungsinstanz im Zustand BARGAINING. Zuerst wartet sie, bis alle Anwendungsinstanzen aus ihrer Clique diesen Zustand erreicht haben. Anschließend beginnt sie mit dem Verhandeln zum Ermitteln der für sie optimalen Anwendungsdomänenkonfiguration, die auch alle Teilnehmer dieser Konfiguration gewählt haben. Danach wechselt die Anwendungsinstanz sofort in den Zustand AUCTION.

```
solution; // Variable für die ausgewählte Anwendungsdomänenkonfiguration

startBargaining()

        solution := null

        wait:=true

        while (wait == true) do

                wait:=false

                forall app from clique do

                        if(app.getApplicationState()<>BARGAINING) do

                                wait:=true

                        end

                end

        end

        forall app from applicationDomainConfigurations.getFirst().getApplications() do

                app.getBargain(applicationDomainConfigurations.getFirst(),this)

        end

        while ( solution == null ) do

        end
```

Abbildung 6-16: Pseudocode zu startBargaining()

In Abbildung 6-16 wird der Pseudocode der Methode „startBargaining“ dargestellt. In der Methode wird zuerst gewartet, bis alle Teilnehmer der Clique sich in dem Zustand BARGAINING befinden. Anschließend sendet jede Anwendungsinstanz ihr bestes Angebot an alle Anwendungsinstanzen, welche an diesem Angebot beteiligt sind. Danach wartet diese Methode, bis eine Lösung gefunden wurde, weil die Anwendungsinstanz nach Beendigung dieser Methode sofort in den Zustand AUCTION wechselt. Die restlichen Aktionen in diesem Zustand finden in den nachfolgenden Methoden statt.

```
bargains; // Variable, in der das aktuelle Angebot jeder anfragenden Anwendungsinstanz gespeichert
werden kann.

getBargain(adc, app)

        if (state == AUCTION && solution.equals(adc) do

                app.getAnswerFromApplication(adc,this,true)

        else if (state <> BARGAINING) do

                app.getAnswerFromApplication(adc,this,false)

        else if (applicationDomainConfigurations.getFirst().equals(adc)) do

                app.getAnswerFromApplication(adc,this,true)

        else if (adc is not in applicationDomainConfigurations) do

                app.getAnswerFromApplication(adc,this,false)

        else do

                bargains(app):=adc

                forall adc2 from applicationDomainConfigurations do

                        if( adc2 < adc && app is in adc2.getApplications()) do

                                applicationDomainConfigurations.remove(adc2)

                        end

                end

        end
```

Abbildung 6-17: Pseudocode zu getBargain()

In Abbildung 6-17 wird der Pseudocode der Methode „getBargain“ dargestellt. Diese Methode erhält ein Angebot eines Konkurrenten und muss es nun, wie in Kapitel 5 beschrieben, prüfen. Dabei werden nacheinander Bedingungen geprüft, bis eine Bedingung erfüllt ist. Die entsprechende Aktion wird anschließend ausgeführt. Wenn sich diese Anwendungsinstanz schon im Zustand AUCTION befindet und das Angebot von einer Anwendungsinstanz ihrer Lösung kommt, wird eine Zustimmung zurückgegeben. Falls diese Methode aufgerufen wird und die Anwendungsinstanz sich in einem anderen Zustand als BARGAINING oder AUCTION befindet, weil sie z. B. nicht mehr am Verfahren teilnimmt, wird das Angebot abgelehnt. Eine Zustimmung erhält der Konkurrent, wenn das Angebot dem aktuellen Angebot dieser Anwendungsinstanz entspricht. Wenn sich das Angebot nicht in der eigenen Liste der Anwendungsdomänenkonfiguration befindet, wird wiederum abgelehnt.

Falls keine dieser Bedingungen erfüllt werden kann, bedeutet dies, dass dieses Angebot sich in der Liste der Anwendungsdomänenkonfigurationen weiter hinten befindet. In dem Fall wird das Angebot für eine spätere Antwort abgespeichert. Dabei wird immer nur das aktuellste Angebot eines Konkurrenten gespeichert. Auf ältere Angebote eines Konkurrenten wird keine Antwort mehr erwartet. Abschließend werden alle Anwendungsdomänenkonfigurationen aus der Liste gelöscht, welche diesen Konkurrenten beinhalten und vor seinem aktuellen Angebot in der sortierten Liste stehen. Aufgrund des aktuellen Angebots weiß die Anwendungsinstanz, dass der Konkurrent diese Anwendungsdomänenkonfigurationen nicht in seiner Liste besitzt.

```
positiveAnswers; //Anwendungsinstanzen, welche auf das aktuelle Angebot positiv geantwortet haben.
getAnswerFromApplication(adc,application,answer)
        if (answer == false && applicationDomainConfigurations.getFirst().equals(adc)) do
                applicationDomainConfigurations.removeFirst()
                positiveAnswers:=
                forall app from applicationDomainConfigurations.getFirst().getApplications() do
                        app.getBargain(applicationDomainConfigurations.getFirst(),this)
                end
                forall app from bargains do
                        if(applicationDomainConfigurations.getFirst() equals bargains(app)) do
                                app.getAnswerFromApplication(bargains(app),this,true)
                                bargains.remove(app)
                        else if (bargains(app)< applicationDomainConfigurations.getFirst()) do
                                app.getAnswerFromApplication(bargains(app),this,false)
                                bargains.remove(app)
                        end
                end
        else if (answer == true && applicationDomainConfigurations.getFirst().equals(adc)) do
                positiveAnswers.add(application)
                test:=true
                forall app from applicationDomainConfigurations.getFirst(),getApplications() do
                        if (app is not in positiveAnswers) do
                                test:=false
                        end
                end
                if (test==true) do
                        solution:= applicationDomainConfigurations.getFirst()
                        state := AUCTION
                        forall app from bargains do
                                app.getAnswerFromApplication(bargains(app),this,false)
                        end
                end
        end
```

Abbildung 6-18: Pseudocode zu getAnswerFromApplication()

In Abbildung 6-18 wird die Antwort auf das aktuelle Angebot geprüft. Bei einer Absage wird die Anwendungsdomänenkonfiguration zusammen mit den bisherigen Zusagen gelöscht und das nächste Angebot aus der Liste an alle beteiligten Anwendungsinstanzen gesendet. Anschließend wird untersucht, ob nun noch unbeantwortete Angebote beantwortet werden können. Bei einer Zusage werden alle bisherigen Zusagen gezählt. Haben alle Anwendungsinstanzen diesem Angebot zugestimmt, wird dieses Angebot gespeichert, der Zustand in AUCTION geändert und anschließend werden alle bisherigen noch unbeantworteten Angebote abgelehnt.

6.2.7 Implementierung Schritt 5

In diesem Schritt befinden sich die Anwendungsinstanzen und Komponenten im Zustand AUCTION. Die Anwendungsinstanzen senden ihre im vierten Schritt ermittelte Anwendungsdomänenkonfiguration an alle daran beteiligten Komponenten. Die übrigen Komponenten aus der Variablen „offers“ erhalten eine Nachricht ohne Anwendungsdomänenkonfiguration. Damit wird gewährleistet, dass jede Komponente von jeder Anwendungsinstanz, der sie im Zustand LISTEN ein Angebot geschickt hat, auch ein Gebot erhält. Anschließend wartet die Anwendungsinstanz, bis sie entweder von einer Komponente eine Ablehnung oder von allen Komponenten eine Zusage erhält. Die Komponente wartet, bis sie alle Gebote erhalten hat, wählt die Anwendungsdomänenkonfiguration mit der höchsten Gewichtung aus und sendet den daran beteiligten Anwendungsinstanzen eine Zustimmung. Die übrigen Anwendungsinstanzen erhalten eine Ablehnung. Wenn mehr als eine Anwendungsdomänenkonfiguration die gleiche höchste Gewichtung besitzt, wird von denen die Anwendungsdomänenkonfiguration mit dem höchsten Hash-Wert gewählt. Somit ist die Wahl immer eindeutig. Abschließend wird die gewählte Anwendungsdomänenkonfiguration umgesetzt.

```
bids; //Menge an Komponenten, denen ein Gebot mit Anwendungsdomänenkonfiguration gesendet wurde.

sendBid()

        forall template from offers do

                forall component from offers(template) do

                        if ( component is in solution.getComponents(this)

                          && component is not in bids) do

                                component.getBid(solution,this)

                                bids.add(component)

                        else if (component is not in solution.getComponents(this)) do

                                component.getBid(null,this)

                        end

                end

        end

        while( state == AUCTION)

        end
```

Abbildung 6-19: Pseudocode zu sendBid()

In Abbildung 6-19 wird der Pseudocode der Methode „sendBid“ dargestellt. In dieser Methode sendet die Anwendungsinstanz allen Komponenten, welche im Zustand SEARCHING ein Angebot gesendet haben, ein Gebot. Die Komponenten, welche sich in der ermittelten Anwendungsdomänenkonfiguration befinden, erhalten diese Anwendungsdomänenkonfiguration als Gebot. Die Übrigen erhalten ein leeres Gebot. Anschließend wartet die Anwendungsinstanz, bis sich ihr Zustand ändert.

```
applicationBids; //Menge der Gebote der Anwendungsinstanzen.

getBid(adc, application)

        if( application is not in applicationBids && application is in applications) do

                applicationBids(application):=adc

        end

        bestSolution := null

        if( applicationBids.size() == applications.size())do

                forall app from applicationBids do

                        if (applicationBids(app) <> null

                            && (bestSolution==null || applicationBids(app) < bestSolution) do

                                bestSolution:=applicationBids(app)

                        end

                end

                if (bestSolution <> null) do

                        forall app from applications do

                                if( app is in bestSolution.getApplications() ) do

                                        app.getAnswerFromComponents(this,true)

                                else

                                        app.getAnswerFromComponents(this,false)

                                end

                        end

                else

                        reset()

                        state := LISTEN

                end

        end
```

Abbildung 6-20: Pseudocode zu getBid()

In Abbildung 6-20 wird der Pseudocode der Methode „getBid“ der Komponenten abgebildet. In dieser Methode speichert die Komponente das erste Gebot jeder der Anwendungsinstanzen,

welchen er im Zustand LISTEN ein Angebot gesendet hat. Ist von all diesen Anwendungsinstanzen ein Gebot eingegangen, wird das Gebot mit der höchsten Gewichtung oder bei gleicher Gewichtung mit dem höchsten Hash-Wert ermittelt. Alle Anwendungsinstanzen dieser Anwendungsdomänenkonfiguration erhalten eine Zusage. Die übrigen Anwendungsinstanzen erhalten auf ihr Gebot eine Absage. Sollte kein gültiges Gebot existieren, wechselt die Komponente wieder in den Zustand LISTEN und löscht alle in dem Verfahren gespeicherten Daten.

```
answerFromBids;  //Komponenten, welche positiv geantwortet haben.

getAnswerFromComponents(component, answer)

        if(answer==false && component is in bids) do

                reset()

                state:= SEARCHING

        else if (answer==true && component is in bids) do

                if (component is not in answerFromBids) do

                        answerFromBids.add(component)

                end

        end

        if(answerFromBids.size() == bids.size()) do

                maxHashValue:= this.hashCode()

                forall app from solution.getApplications() do

                        maxHashValue:= MAX(maxHashValue, app.hashCode())

                end

                if (maxHashValue==this.hashCode()) do

                        solution.createRealConfiguration()

                end

                state:=RUNNING

        end
```

Abbildung 6-21: Pseudocode zu getAnswerFromComponents()

In Abbildung 6-21 wird der Pseudocode zum Bearbeiten der Antworten der Komponenten dargestellt. Bei einer negativen Antwort einer der benötigten Komponenten wechselt die Anwendungsinstanz in den Zustand SEARCHING und löscht alle im Lösungsverfahren gespeicherten Daten. Wenn alle angefragten Komponenten zusagen, erstellt die Anwendungsinstanz mit dem höchsten Hash-Wert aus der Anwendungsdomänenkonfiguration die Verschaltung. Anschließend wechselt sie in den Zustand RUNNING.

Damit wurde die Implementierung jedes Schrittes des Lösungsverfahrens beschrieben. Im nächsten Kapitel folgt eine Beschreibung zur Verwendung des Prototyps. Anschließend wird die Implementierung der Anwendungen und Komponenten des Anwendungsbeispiels beschrieben und das Verfahren an den Szenarien evaluiert.

7 Evaluierung

Im vorherigen Kapitel wurde eine prototypische Implementierung des Metamodells aus Kapitel 4 vorgestellt. Dieser Prototyp verschaltet Komponenten untereinander zu Anwendungsinstanzen unter Verwendung des Verfahrens, welches in Kapitel 5 beschrieben wurde.

In diesem Kapitel wird das Verfahren mittels des Prototyps aus Kapitel 6 prototypisch evaluiert. Dabei werden die fünf Szenarien des Anwendungsbeispiels aus Kapitel 3 unter Verwendung des Prototyps implementiert. Anschließend berechnet der Prototyp mit dem Verfahren aus Kapitel 5 die jeweilige Zielsystemkonfiguration. Diese wird mit der in Kapitel 3 beschriebenen Zielsystemkonfigurationen der Szenarien des Anwendungsbeispiels verglichen.

Dieses Kapitel beginnt mit einer Beschreibung einer Implementierung von Komponenten und Anwendungen und beschreibt danach die Komponenten und Anwendungen des Anwendungsbeispiels aus Kapitel 3. Anschließend werden die einzelnen Testfälle durchgeführt und mit den erwarteten Ergebnissen der jeweiligen Szenarien verglichen. Zusätzlich werden noch 2 weitere Testfallszenarien vorgestellt, welche zeigen, dass das Verfahren nicht nur auf das beschriebene Anwendungsbeispiel beschränkt ist.

7.1 Komponentenmodell des Prototyps

Das in Abbildung 7-1 dargestellte UML-Klassendiagramm zeigt die einzelnen Klassen und Schnittstellen des Prototyps. Dabei werden nur die Methoden dargestellt, welche für die Verwendung des Prototyps von Bedeutung sind.

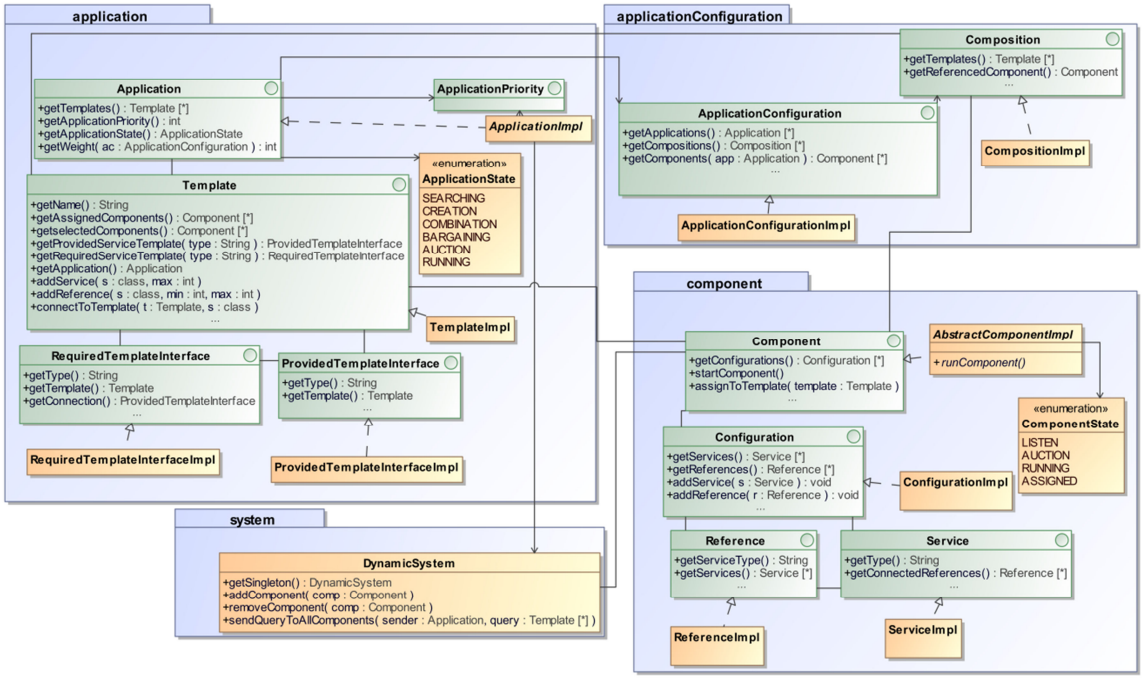

Abbildung 7-1: Klassendiagramm des Prototyps

Über die dargestellten Methoden kann zwischen den Klassen navigiert werden. Damit können zum einen die Gewichtung und zum anderen die Bedingungen aus appConstraint beschrieben werden.

7.2 Implementierung einer Komponente

In Abbildung 7-2 wird eine einfache Beispielkomponente entsprechend der grafischen Notation aus Kapitel 3 dargestellt. Die Komponente besitzt genau eine Konfiguration, in der sie einen Dienst über die Schnittstelle „BeispielService“ anbietet und maximal einen Dienst über die Schnittstelle „BeispielReferenz“ benötigt.

Abbildung 7-2: Grafische Notation der Beispielkomponente

Der Aufbau der Beispielkomponente wird in Abbildung 7-3 in einem Klassendiagramm dargestellt. Darin werden auch die Assoziationen zu den Klassen des Prototyps angezeigt.

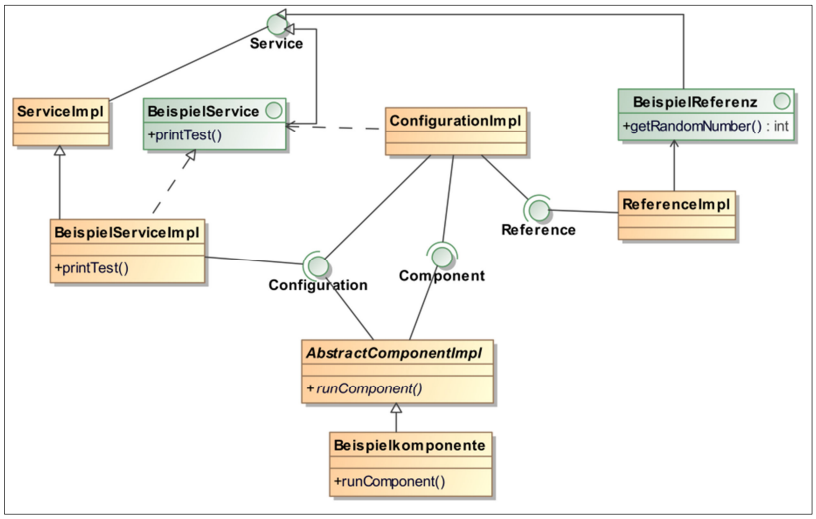

Abbildung 7-3: Klassendiagramm der Beispielkomponente

Für die Beispielkomponente müssen die beiden Schnittstellen „BeispielService“ und „BeispielReferenz“ definiert sowie die beiden Klassen „BeispielServiceImpl“ und „Beispielkomponente“ implementiert werden. Die Klasse „BeispielServiceImpl“ implementiert dabei die Schnittstelle „BeispielService“ zur Bereitstellung des zugehörigen Dienstes. Außerdem wird die Klasse von der Klasse „ServiceImpl“ abgeleitet. Damit erhält die Java-Klasse „BeispielServiceImpl“ den Programmcode zu den in der Schnittstelle Service vorhandenen Methoden.

Die Klasse „Beispielkomponente“ wird von der abstrakten Klasse „AbstractComponentImpl“ abgeleitet und implementiert die Methode „runComponent“. Mit dieser Methode kann die Komponente im Zustand RUNNING eigenständig Aufgaben einer Anwendung ausführen. Da diese Komponente aber nur auf Anfragen reagieren soll, bleibt diese Methode leer. Die Erstellung der Struktur, wie sie in Abbildung 7-3 dargestellt ist, wird im Konstruktor implementiert, wie in Abbildung 7-4 dargestellt.

```
public class Beispielkomponente extends AbstractComponentImpl {

    private BeispielReferenz reference=null;

    public Beispielkomponente(){
        super("Beispielkomponente");

    ConfigurationImpl configuration=createConfiguration();

        Reference r=createReference("reference",0,1);

        configuration.addService(new BeispielServiceImpl());
        configuration.addReference(r);
        addConfiguration(configuration);

        shouldRun=false;
    }


    @Override
    protected void runComponent() {
    }

}
```

Abbildung 7-4: Programmcode der Klasse „Beispielkomponente“

Im Konstruktor werden neben dem Namen der Komponente die Dienste, entweder als angebotener oder benötigter Dienst, und eine Konfiguration erstellt. Für einen benötigten Dienst wird in der Klasse ein Attribut vom Typ der Schnittstelle benötigt. In dem Beispiel ist dies das Attribut „reference“ vom Typ „BeispielReferenz“. Außerdem wird beim Erstellen einer Referenz in der Methode „createReference“ noch die Anzahl der benötigten Dienste angegeben. Nach Verschaltung kann über dieses Attribut auf den benötigten Dienst zugegriffen werden.

Nach der Erstellung der Dienste werden sie einer Konfiguration und die Konfiguration wird anschließend der Klasse „Beispielkomponente“ zugeordnet. Die Variable „shouldRun“ wird benötigt, damit die Komponente weiß, ob sie im Zustand RUNNING selbstständig den Programmcode in der Methode „runComponent“ ausführen soll. Die Variable wird auf „false“ gesetzt, da die Komponente keine eigenständigen Aufgaben besitzt, sondern, wie in dem Anwendungsbeispiel in Kapitel 3 beschrieben, nur einen Dienst für andere Komponenten anbietet. Aus diesem Grund bleibt auch die Methode „runComponent“ leer.

7.3 Implementierung einer Anwendung

Nach Beschreibung einer Beispielkomponente wird die Erstellung einer Anwendung für den Prototyp anhand eines Beispiels erläutert. Die Beispielanwendung ist in Abbildung 7-5 dargestellt.

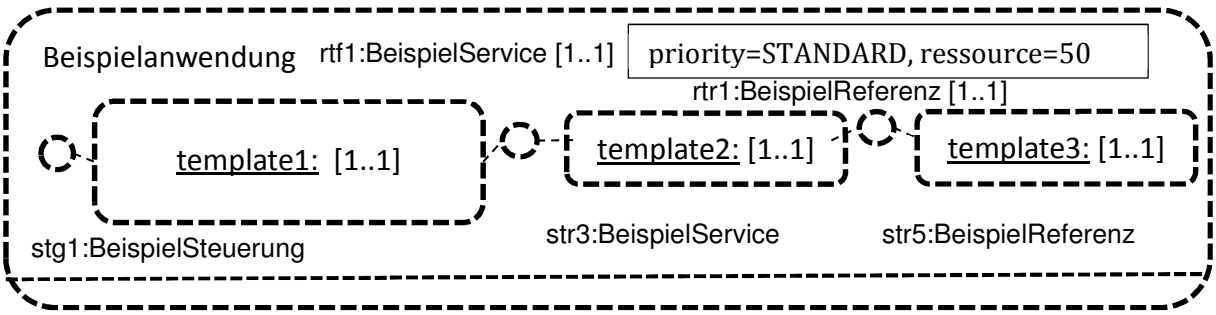

Abbildung 7-5: Grafische Notation der Beispielanwendung

Die Beispielanwendung enthält drei Schablonen, die Priorität STANDARD und einen Ressourcenwert 50, wenn die zugehörige Anwendungsinstanz verschaltet ist. Weitere Bedingungen für eine erfolgreiche Verschaltung besitzt diese Anwendung nicht. In Abbildung 7-6 werden die gleichen Informationen aus der grafischen Notation aus Abbildung 7-5 als Programmcode dargestellt.

```
public class Beispielanwendung extends ApplicationImpl {

    protected Beispielanwendung() {
        super(AirportPriority.STANDARD,"BeispielAnwendung");

        Template t1=createTemplate("template1", 1, 1);
        Template t2=createTemplate("template2", 1, 1);
        Template t3=createTemplate("template3", 1, 1);

        t1.addService(Beispielsteuerung.class, 1);
        t1.addReference(BeispielService.class, 1, 1);

        t2.addService(BeispielService.class, 1);
        t2.addReference(BeispielReferenz.class, 1, 1);

        t3.addService(BeispielReferenz.class,1);


        t1.connectToTemplate(t2, BeispielService.class);
        t2.connectToTemplate(t3, BeispielReferenz.class);

    }

    @Override
    public int getWeight(ApplicationConfiguration ac) {
        if(ac.getApplications().contains(this))
            return 50;
        return 0;
    }

    @Override
    protected boolean appConstraint(ApplicationConfiguration ac) {
        return true;
    }

}
```

Abbildung 7-6: Programmcode der Klasse „Beispielanwendung“

Der erste Befehl setzt die Priorität dieser Anwendung auf STANDARD und gibt der Anwendung einen Namen, was eine spätere Identifizierung im Szenario erleichtert. Im Anschluss werden die drei Schablonen der Beispielanwendung mit den zugehörigen Kardinalitäten erstellt. Danach

werden die Schablonen mit den anbietenden und benötigten Diensten versehen, welche die jeweilige Komponente besitzen muss, um einer der Schablonen zugeordnet werden zu können. Die letzten beiden Befehle verknüpfen die Schablonen miteinander über die jeweilige Schnittstelle. In der Methode „getWeight“ prüft die Beispielanwendung, ob sie Teil der übergebenen Gruppe von Anwendungskonfigurationen ist, und gibt bei Bestätigung den Wert 50 und andernfalls den Wert 0 zurück. Da die Beispielanwendung keine weiteren Bedingungen besitzt, bleibt die Methode „appConstraint“ leer. Es wird lediglich der Wert „true“ auf die Anfrage zurückgeliefert, ob alle Bedingungen erfüllt sind.

In diesem Kapitel wurde ein Prototyp als Implementierung des Verfahrens vorgestellt und dessen Nutzung anhand einer Beispielkomponente und einer Beispielanwendung erläutert. Im nächsten Kapitel werden die Implementierungen der Anwendungen und Komponenten aus dem Anwendungsbeispiel aus Kapitel 3 beschrieben. Anschließend folgt ein Vergleich der zu erwartenden Ergebnisse der Szenarien des Anwendungsbeispiels mit den Ergebnissen der Verschaltung des Prototyps bei den einzelnen Szenarien.

7.4 Komponenten

Nachfolgend werden die Implementierungen der einzelnen Komponenten und Anwendungen des Anwendungsbeispiels aus Kapitel 3 kurz dargestellt und beschrieben. Dabei wird der Programmcode, welcher im Zustand RUNNING ausgeführt wird, ausgeblendet, weil dieser für die Evaluation des Verfahrens nicht relevant ist.

```
public class RoboterImpl extends AbstractComponentImpl {

    private RoboterkolonneI roboter = null;

    public RoboterImpl(String name) {
        super(name);

        ConfigurationImpl best = createConfiguration();

        Reference r = createReference("roboter", 0, 1);

        best.addService(new RoboterServiceImpl());
        best.addService(new RoboterkolonneImpl());
        best.addReference(r);
        addConfiguration(best);

        shouldRun = false;

    }

    @Override
    protected void runComponent() {
    }


}
```

Abbildung 7-7: Ausschnitt aus der Implementierung einer Roboterkomponente

In Abbildung 7-7 wird der für die Verschaltung des Verfahrens relevante Programmcode der Roboterkomponente dargestellt. Wie schon in Abbildung 3-13 beschrieben, bietet die Komponente die Dienste „RoboterI“ und „RoboterkolonneI“ an, welche jeweils in den Klassen „Roboter-

ServiceImpl“ und „RoboterkolonneImpl“ implementiert sind. Des Weiteren benötigt die Implementierung der Roboterkomponente einen Dienst vom Typ „RoboterkolonneI“, welcher in Abbildung 7-7 über die Methode „createReference“ beschrieben ist. Die Variable „shouldRun“ zeigt an, dass die Komponente im Zustand RUNNING keine eigenen Aufgaben zu bewältigen hat.

```java
public class RoboterGUIImpl extends AbstractComponentImpl {

    private RoboterI roboter = null;
    private String ziel = null;

    public RoboterGUIImpl(String ziel,String name) {
        super(name);
        this.ziel = ziel;
        ConfigurationImpl best = createConfiguration();

        Reference r = createReference("roboter", 1, 1);

        best.addService(new RoboterGUIServiceImpl(ziel));
        best.addReference(r);
        addConfiguration(best);

        shouldRun = true;
    }

    @Override
    protected void runComponent() {
     ...

    }

}
```

```java
public class KolonnensteruerungImpl extends AbstractComponentImpl {

    private RoboterI roboter=null;
    private String ziel=null;

    public KolonnensteruerungImpl(String ziel){
        super("Kolonnensteuerung");
        this.ziel=ziel;
        ConfigurationImpl best = createConfiguration();

        Reference r = createReference("roboter", 1, 1);

        best.addService(new KolonnensteuerungServiceImpl(ziel));
        best.addReference(r);
        addConfiguration(best);

        shouldRun = true;
    }

    @Override
    protected void runComponent() {
     ...


    }

}
```

Abbildung 7-8: Ausschnitt aus der Implementierung der beiden Komponenten RoboterGUI und Kolonnensteuerung

In Abbildung 7-8 wird die Implementierung der Komponenten „RoboterGUI“ und „Kolonnensteuerung“ dargestellt, welche jeweils einen Dienst anbieten, der die Schnittstelle „HasZielI“ implementiert (siehe Abbildung 3-14). Daher erhalten beide Komponenten ein Ziel, welches sie an den jeweiligen Dienst weiterleiten. Außerdem erstellen beide Komponenten eine Referenz auf einen benötigten Dienst vom Typ „RoboterI“. Beide Komponenten setzen die Variable „shouldRun“ auf true, weil sie Programmcode besitzen, welcher im Zustand RUNNING ausgeführt werden soll. Dieser Code befindet sich in der Methode „runComponent“, welcher hier aber nicht abgebildet wird, da die Funktionen der beiden Komponenten für diese Arbeit nicht relevant sind.

```
public class FluchtwegsicherungImpl extends AbstractComponentImpl {

    private LinkedList<RoboterI> roboter=new LinkedList<>();

    public FluchtwegsicherungImpl(){
        super("Fluchtwegsicherung");
        ConfigurationImpl best = createConfiguration();

        Reference r = createReference("roboter", 1, 0);

        best.addService(new FluchtwegsicherungServiceImpl());
        best.addReference(r);
        addConfiguration(best);

        shouldRun = true;
    }

    @Override
    protected void runComponent() {
    ...

    }

}
```

Abbildung 7-9: Ausschnitt aus der Implementierung einer Fluchtwegsicherungskomponente

In Abbildung 7-9 wird die Komponente „Fluchtwegsicherung“, wie in Abbildung 3-15 beschrieben, dargestellt. Sie unterscheidet sich von den beiden eben beschriebenen Komponenten „RoboterGUI“ und „Kolonnensteuerung“ in ihrem Aufbau nur in einer Tatsache: Sie kann sich über einen Dienst mit der Schnittstelle „RoboterI“ mit einer Menge von Komponenten verbinden.

7.5 Anwendungen

Die Anwendungen des Anwendungsbeispiels aus Kapitel 3 sind analog zur Beispielanwendung aus Kapitel 6 aufgebaut. Sie besitzen einen Konstruktor, welcher die Struktur der Anwendung erstellt. Des Weiteren besitzt die Anwendung eine Methode „getWeight“ zur Bestimmung der Gewichtung und eine Methode „appConstraint“, in der die Bedingungen der Anwendungen, wie in Kapitel 3 beschrieben, enthalten sind.

```
public class VIP extends ApplicationImpl {

    public VIP(String name) {
        super(AirportPriority.STANDARD,name);

        Template tRoboterGUI=createTemplate("tRoboterGUI", 1, 1);
        Template tRoboter=createTemplate("tRoboter1", 1, 1);

        tRoboterGUI.addService(RoboterGUII.class, 1);
        tRoboterGUI.addReference(RoboterI.class, 1, 1);

        tRoboter.addService(RoboterI.class, 1);

        tRoboterGUI.connectToTemplate(tRoboter, RoboterI.class);

        RoboterGUIImpl g=new RoboterGUIImpl(null,"RoboterGUI von "+name);
        tRoboterGUI.assignComponent(g);

    }


    @Override
    public int getWeight(ApplicationConfiguration ac) {
        if(ac.getApplications().contains(this))
            return 100;
        return 0;
    }

    @Override
    protected boolean appConstraint(ApplicationConfiguration ac) {
        LinkedList<Composition> cc=ac.getCompositions();
        Template t=null;

        for (Template tt:this.getTemplates()){
            if(tt.getName().equals("tRoboter1"))
                t=tt;
        }

        for(Composition c:cc){
            if(c.getTemplates().contains(t) && c.getTemplates().size()>1)
                return false;
        }

        return true;
    }

}
```

Abbildung 7-10: Ausschnitt aus der Implementierung der Anwendung VIP

In Abbildung 7-10 wird die Implementierung der VIP-Anwendung dargestellt. Wie in Abbildung 3-17 zu sehen, besitzt die Anwendung zwei Schablonen, wobei eine Schablone einen anbietenden Dienst vom Typ „RoboterGUII“ und einen benötigten Dienst vom Typ „RoboterI“ besitzt. Die andere Schablone besitzt einen anbietenden Dienst vom Typ „RoboterI“. Außerdem wird eine Komponente „RoboterGUI“ mit der Anwendung gestartet und der ersten Schablone fest zugeordnet. Die Methode „getWeight“ liefert die Gewichtung 100 der VIP-Anwendung, wenn eine Anwendungskonfiguration dieser Anwendung in der übergebenen Anwendungsdomänenkonfiguration (ApplicationConfiguration) enthalten ist. Die Methode „appConstraint“ prüft, ob die Komponente, welche in der Anwendungsdomänenkonfiguration der Schablone „tRoboter1“ die-

ser VIP-Anwendung zugeordnet ist, auch keinem Template einer weiteren Anwendung zugeordnet ist.

```
public class Kunde extends ApplicationImpl {

    public Kunde(String ziel,String name) {
        super(AirportPriority.STANDARD,name);

        Template tRoboterGUI=createTemplate("tRoboterGUI", 1, 1);
        Template tRoboter=createTemplate("tRoboter1", 1, 1);

        tRoboterGUI.addService(RoboterGUII.class, 1);
        tRoboterGUI.addReference(RoboterI.class, 1, 1);

        tRoboter.addService(RoboterI.class, 1);

        tRoboterGUI.connectToTemplate(tRoboter, RoboterI.class);

        RoboterGUIImpl g=new RoboterGUIImpl(ziel,"RoboterGUI von "+name);
        tRoboterGUI.assignComponent(g);
    }

    @Override
    public int getWeight(ApplicationConfiguration ac) {
        if(ac.getApplications().contains(this))
            return 60;
        return 0;
    }

    @Override
    protected boolean appConstraint(ApplicationConfiguration ac) {
        LinkedList<Composition> cc=ac.getCompositions();
        Template t=null;
        String zeal=null;
        for (Template tt:this.getTemplates()){
            if(tt.getName().equals("tRoboter1"))
                t=tt;
            if(tt.getName().equals("tRoboterGUI")){
                for(Configuration c:tt.getAssignedComponents().get(0).getConfigurations()){
                    for(Service s:c.getServices()){
                        if( s instanceof HasZielI){
                            zeal=((HasZielI)s).getZiel();
                        }
                    }
                }
            }
        }

        Composition comp=null;
        for(Composition c:cc){
            if(c.getTemplates().contains(t))
                comp=c;
        }

        for(Template t2:comp.getTemplates()){
            if(t2.getProvidedServiceTemplate(RoboterI.class)==null)
                continue;
            Template temp=t2.getProvidedServiceTemplate(RoboterI.class).getConnection().getTemplate();
            for(Composition c:cc){
                if(c.getTemplates().contains(temp)){

                    for(Configuration conf:c.getReferencedComponent().getConfigurations()){
                        for(Service s:conf.getServices()){
                            if( s instanceof HasZielI){
                                if(((HasZielI)s).getZiel()!=null && !((HasZielI)s).getZiel().equals(zeal)){
                                    return false;
                                }
                            }
                        }
                    }
                }
            }
        }

        return true;
    }

}
```

Abbildung 7-11: Ausschnitt aus der Implementierung der Anwendung Kunde

Die Implementierung der Anwendung „Kunde“ ist in Abbildung 7-11 beschrieben. Ihre Struktur entspricht der Beschreibung der Anwendung aus Abbildung 3-16. Wie auch bei der VIP-Anwendung wird mit dem Starten einer Kundenanwendung eine Komponente „RoboterGUI“ gestartet und der entsprechenden Schablone dieser Anwendung fest zugeordnet. Die Methode „getWeight“ liefert die Gewichtung 60, wenn diese Kundenanwendung eine Anwendungskonfiguration in der übergebenen Anwendungsdomänenkonfiguration enthält. In der Methode „appConstraint“ wird, wie in Abbildung 3-16 beschrieben, geprüft, ob die dieser Anwendung zugeordnete Roboterkomponente nur mit Anwendungen verbunden ist, welche alle, wenn vorhanden, das gleiche Ziel wie diese Anwendung besitzen.

```
public class Gepäckwagentransport extends ApplicationImpl {

    public Gepäckwagentransport(String ziel) {

        super(AirportPriority.STANDARD,"Gepäcktwagenransport");

        Template tKolonnensteuerung=createTemplate("tKolonnensteuerung", 1, 1);
        Template tRoboter1=createTemplate("tRoboter1", 1, 1);
        Template tRoboter2=createTemplate("tRoboter2", 1, 1);

        tKolonnensteuerung.addService(KolonnensteuerungI.class, 1);
        tKolonnensteuerung.addReference(RoboterI.class, 1, 1);

        tRoboter1.addService(RoboterI.class, 1);
        tRoboter1.addReference(RoboterkolonneI.class, 1, 1);

        tRoboter2.addService(RoboterkolonneI.class, 1);

        tKolonnensteuerung.connectToTemplate(tRoboter1, RoboterI.class);
        tRoboter1.connectToTemplate(tRoboter2, RoboterkolonneI.class);

        KolonnensteruerungImpl g=new KolonnensteruerungImpl(ziel);
        tKolonnensteuerung.assignComponent(g);
    }

    @Override
    public int getWeight(ApplicationConfiguration ac) {
        if(ac.getApplications().contains(this))
            return 150;
        return 0;
    }
```

Abbildung 7-12: Ausschnitt aus der Implementierung der Anwendung Gepäckwagentransport Teil 1

```
    @Override
    protected boolean appConstraint(ApplicationConfiguration ac) {
        LinkedList<Composition> cc=ac.getCompositions();
        Template t1=null,t2=null;
        String zeal=null;
        for (Template tt:this.getTemplates()){
            if(tt.getName().equals("tRoboter1"))
                t1=tt;
            if(tt.getName().equals("tRoboter2"))
                t2=tt;
            if(tt.getName().equals("tKolonnensteuerung")){
                for(Configuration c:tt.getAssignedComponents().get(0).getConfigurations()){
                    for(Service s:c.getServices()){
                        if( s instanceof HasZielI){
                            zeal=((HasZielI)s).getZiel();
                        }
                    }
                }
            }
        }
        Composition comp1=null,comp2=null;
        for(Composition c:cc){
            if(c.getTemplates().contains(t1))
                comp1=c;
            if(c.getTemplates().contains(t2))
                comp2=c;
        }

        for(Template tt2:comp1.getTemplates()){
            if(tt2.getProvidedServiceTemplate(RoboterI.class)==null)
                continue;
            Template temp=tt2.getProvidedServiceTemplate(RoboterI.class).getConnection().getTemplate();
            for(Composition c:cc){
                if(c.getTemplates().contains(temp)){

                    for(Configuration conf:c.getReferencedComponent().getConfigurations()){
                        for(Service s:conf.getServices()){
                            if( s instanceof HasZielI){
                                if(((HasZielI)s).getZiel()!=null && !((HasZielI)s).getZiel().equals(zeal)){
                                    return false;
                                }
                            }
                        }
                    }
                }
            }
        }
        for(Template tt2:comp2.getTemplates()){
            if(tt2.getProvidedServiceTemplate(RoboterI.class)==null)
                continue;
            Template temp=tt2.getProvidedServiceTemplate(RoboterI.class).getConnection().getTemplate();
            for(Composition c:cc){
                if(c.getTemplates().contains(temp)){

                    for(Configuration conf:c.getReferencedComponent().getConfigurations()){
                        for(Service s:conf.getServices()){
                            if( s instanceof HasZielI){
                                if(((HasZielI)s).getZiel()!=null && !((HasZielI)s).getZiel().equals(zeal)){
                                    return false;
                                }
                            }
                        }
                    }
                }
            }
        }
        return true;
    }
}
```

Abbildung 7-13: Ausschnitt aus der Implementierung der Anwendung Gepäckwagentransport Teil 2

Die Implementierung der Anwendung „Gepäckwagentransport“ in Abbildung 7-12 entspricht in ihrer Struktur ebenfalls der Beschreibung aus Abbildung 3-18. Analog zu den anderen Anwendungen wird beim Starten einer Gepäckwagentransportanwendung eine Komponente „Kolonnensteuerung“ gestartet und der entsprechenden Schablone dieser Anwendung fest zugeordnet. Die Methode „getWeight“ liefert die Gewichtung 150, wenn die Gepäckwagentransportanwendung eine Anwendungskonfiguration der übergebenen Anwendungsdomänenkonfiguration bei-

steuert. In der Methode „appConstraint“ in Abbildung 7-13 wird geprüft, ob beide dieser Anwendung zugeordneten Roboterkomponenten nur mit Anwendungen verknüpft sind, die, wenn vorhanden, das gleiche Ziel wie diese Gepäckwagentransportanwendung besitzen.

```
public class Evakuierung extends ApplicationImpl {

    public Evakuierung() {
        super(AirportPriority.EMERGENCY,"Evakuierung");

        Template tFluchtwegsicherung=createTemplate("tFluchtwegsicherung", 1, 1);
        Template tRoboter=createTemplate("tFRoboter1", 1, 0);

        tFluchtwegsicherung.addService(FluchtwegsicherungI.class, 1);
        tFluchtwegsicherung.addReference(RoboterI.class, 1, 0);

        tRoboter.addService(RoboterI.class, 1);

        tFluchtwegsicherung.connectToTemplate(tRoboter, RoboterI.class);

        FluchtwegsicherungImpl g=new FluchtwegsicherungImpl();
        tFluchtwegsicherung.assignComponent(g);
    }

    @Override
    public int getWeight(ApplicationConfiguration ac) {
        int out=0;
        if(ac.getApplications().contains(this))
            out=(ac.getComponents(this).size()-ac.getAssignedComponents(this).size())*10;
        return out;
    }

    @Override
    protected boolean appConstraint(ApplicationConfiguration ac) {
        return true;
    }

}
```

Abbildung 7-14: Ausschnitt aus der Implementierung der Anwendung Evakuierung

In Abbildung 7-14 wird die Implementierung der Anwendung „Evakuierung“ abgebildet. Die Befehle im Konstruktor der Anwendung entsprechen der Anwendungsbeschreibung in Abbildung 3-19. Außerdem wird bei Start einer Evakuierungsanwendung eine Fluchtwegsicherungskomponente gestartet und der entsprechenden Schablone dieser Anwendung fest zugeordnet. Die Methode „getWeight“ liefert als Gewichtung die Summe der Roboterkomponenten um den Faktor 10 erhöht. Da diese Anwendung keine weiteren Bedingungen für eine Anwendungskonfiguration besitzt, bleibt die Methode „appConstraint“ leer.

7.6 Testfälle

Mit der prototypischen Ausführung der Szenarien des Anwendungsbeispiels in den folgenden Testfällen soll gezeigt werden, dass das Verfahren aus Kapitel 5 eine Lösung der aus dem Anwendungsbeispiel resultierenden Problemstellung aus Kapitel 3 darstellt. Dafür werden die in Kapitel 7.1 beschriebenen Komponenten und die in Kapitel 7.2 beschriebenen Anwendungen verwendet.

Jeder Testfall beginnt mit einer kurzen Beschreibung des zugehörigen Szenarios. Anschließend zeigt eine Abbildung die Ausgangssystemkonfiguration und Zielsystemkonfiguration des Testfalls. Die Zielsystemkonfiguration des Testfalls wird danach mit dem erwarteten Ergebnis des jeweiligen Szenarios, welches in Kapitel 3 beschrieben wurde, verglichen.

Abbildung 7-15: Grafische Darstellung einer Systemkonfiguration im Prototyp

In Abbildung 7-15 wird eine Systemkonfiguration aus dem Prototyp dargestellt. Diese Darstellung wird in den nachfolgenden Testfällen verwendet. Die linke Spalte der tabellarischen Darstellung enthält die Namen der Komponenten, Anwendungen und Schablonen der Anwendungen und die rechte Spalte den jeweiligen Zustand der Komponente oder Anwendung. Die ersten Zeilen der Tabelle enthalten alle nicht fest zugewiesenen Komponenten, welche in dem System aktiv sind. Jeweils nach einer freien Zeile wird eine Anwendung zusammen mit den enthaltenen Schablonen und den Komponenten, welche den jeweiligen Schablonen zugeordnet sind, dargestellt.

Die Systemkonfiguration in Abbildung 7-15 enthält eine nicht fest zugewiesene Komponente mit Namen „Roboter“ und zwei Anwendungen, „Kunde“ und „vip“. Die Komponente „Roboter“ befindet sich im Zustand RUNNING und ist der Anwendungen „vip“ zugeordnet, welche sich ebenfalls im Zustand RUNNING befindet. Die Anwendung „Kunde“ befindet sich im Zustand SEARCHING und enthält eine fest zugeordnete Komponente „RoboterGUI von Kunde“, welche sich im Zustand ASSIGNED befindet.

7.6.1 Testfall 1

Der erste Testfall führt Szenario 1 des Anwendungsbeispiels aus Kapitel 3 durch. Die Ausgangssystemkonfiguration und erwartete Zielsystemkonfiguration sind in Abbildung 3-20 dargestellt.

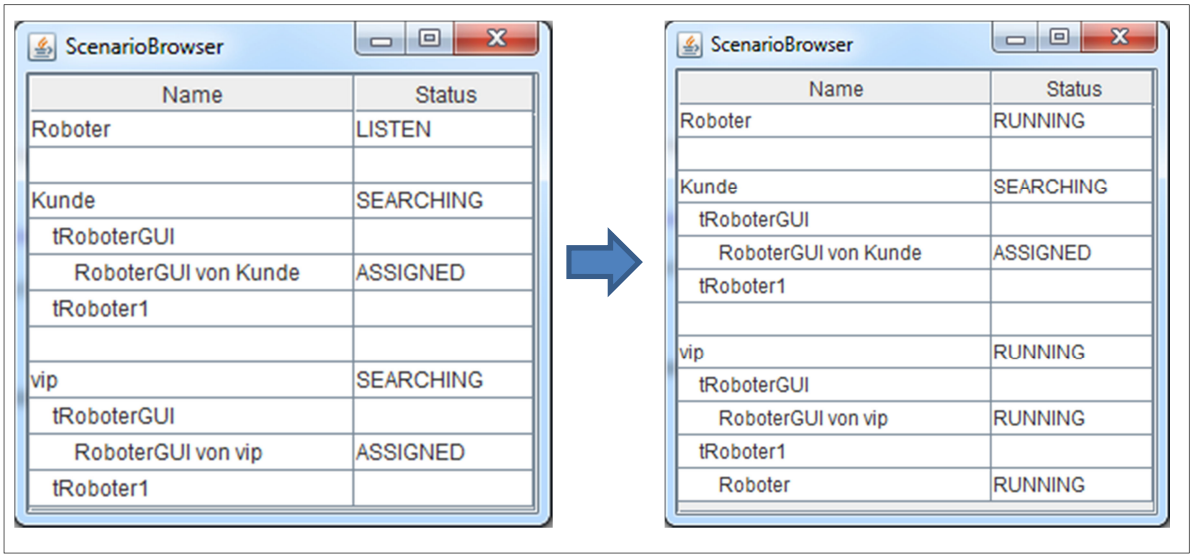

Abbildung 7-16: Testfall 1

Abbildung 7-16 ist analog zu Abbildung 3-20 aufgebaut. Auch hier werden zwei Systemkonfigurationen dargestellt, welche mit einem blauen Pfeil verbunden sind. Die linke Systemkonfiguration stellt die Ausgangssystemkonfiguration vor Durchführung des implementierten Verfahrens dar. Die rechte Systemkonfiguration zeigt die Zielsystemkonfiguration nach Durchführung des implementierten Verfahrens. Die Ausgangssystemkonfiguration enthält, wie in Szenario 1 des Anwendungsbeispiels beschrieben, eine freie Komponente vom Typ „Roboter“ und die zwei Anwendungen „Kunde“ und „vip“. Nach Durchführung des Verfahrens ist die Komponenten „Roboter“ der Anwendung „vip“ zugeordnet und befindet sich wie auch die Anwendung „vip“ im Zustand RUNNING. Die Anwendung „Kunde“ befindet sich weiterhin im Zustand SEARCHING.

Die Zielsystemkonfiguration aus Abbildung 7-16 entspricht der Zielsystemkonfiguration aus Abbildung 3-20 des 1. Szenarios des Anwendungsbeispiels. Das implementierte Verfahren erfüllt demnach alle Anforderungen des 1. Szenarios des Anwendungsbeispiels.

7.6.2 Testfall 2

Im zweiten Testfall wird Szenario 2 des Anwendungsbeispiels aus Kapitel 3 geprüft. Die Ausgangssystemkonfiguration und erwartete Zielsystemkonfiguration sind in Abbildung 3-21 dargestellt.

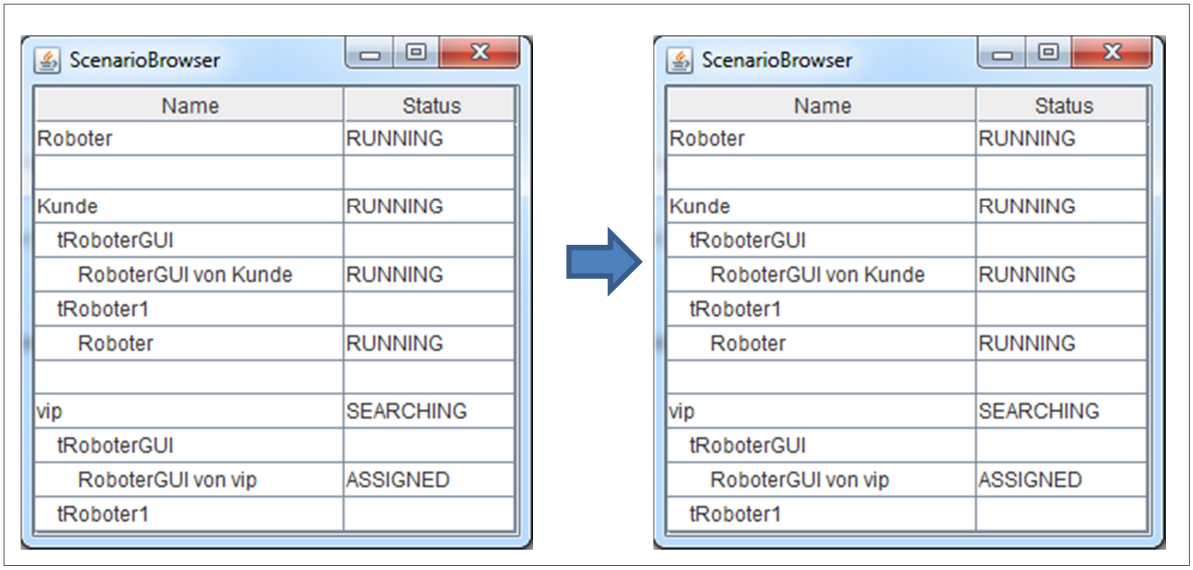

Abbildung 7-17: Testfall 2

In Abbildung 7-17 wird die Ausgangssystemkonfiguration von Abbildung 3-21 dargestellt. Die Ausgangssystemkonfiguration enthält eine Komponente „Roboter“ und die zwei Anwendungen „Kunde“ und „vip“, wobei die Komponente „Roboter“ der Anwendung „Kunde“ zugeordnet ist. Sowohl die Komponente „Roboter“ als auch die Anwendung „Kunde“ befinden sich im Zustand RUNNING. Die Anwendung „vip“ befindet sich im Zustand SEARCHING. Die Zielsystemkonfiguration nach der Anwendung des implementierten Verfahrens gleicht bei diesem Testfall der Ausgangssystemkonfiguration. Damit entspricht die Zielsystemkonfiguration aus Abbildung 7-17 genau der Zielsystemkonfiguration aus dem 2. Szenario des Anwendungsbeispiels, siehe Abbildung 3-21. Das implementierte Verfahren erfüllt demnach auch die Anforderung aus dem 2. Szenario.

7.6.3 Testfall 3

Im dritten Testfall wird Szenario 3 des Anwendungsbeispiels aus Kapitel 3 ausgeführt. In Abbildung 3-22 sind die Ausgangssystemkonfiguration und die erwartete Zielsystemkonfiguration für diesen Testfall dargestellt.

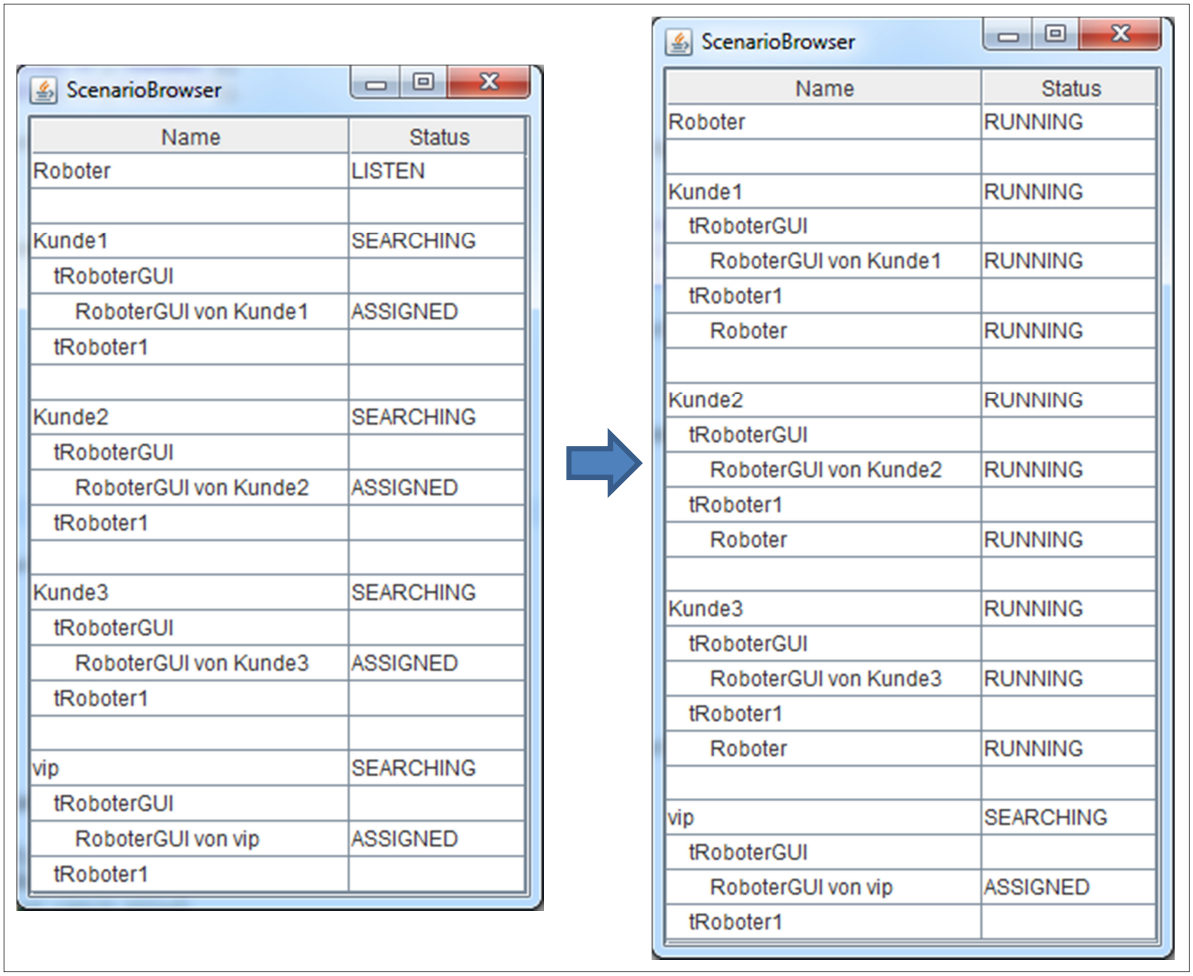

Abbildung 7-18: Testfall 3

Die Ausgangssystemkonfiguration in Abbildung 7-18 enthält eine freie Komponente „Roboter“ und die vier Anwendungen „Kunde1“, „Kunde2“, „Kunde3“ und „vip“, wie auch in Abbildung 3-22 im 3. Szenario beschrieben. In der Zielsystemkonfiguration in Abbildung 7-18 ist die Komponente „Roboter“ den drei Anwendungen „Kunde1“, „Kunde2“ und „Kunde3“ zugeordnet. Diese drei Anwendungen und die Komponente „Roboter“ befinden sich im Zustand RUNNING. Die Anwendung „vip“ befindet sich weiterhin im Zustand SEARCHING.

Auch in diesem Testfall entspricht die Zielsystemkonfiguration genau der Zielsystemkonfiguration aus dem zugehörigen Szenario. Das implementierte Verfahren erfüllt demnach ebenfalls die Anforderung aus dem 3. Szenario.

7.6.4 Testfall 4

Szenario 4 des Anwendungsbeispiels wird im vierten Testfall untersucht. Die Ausgangssystemkonfiguration und die erwartete Zielsystemkonfiguration für diesen Testfall sind in Abbildung 3-23 dargestellt.

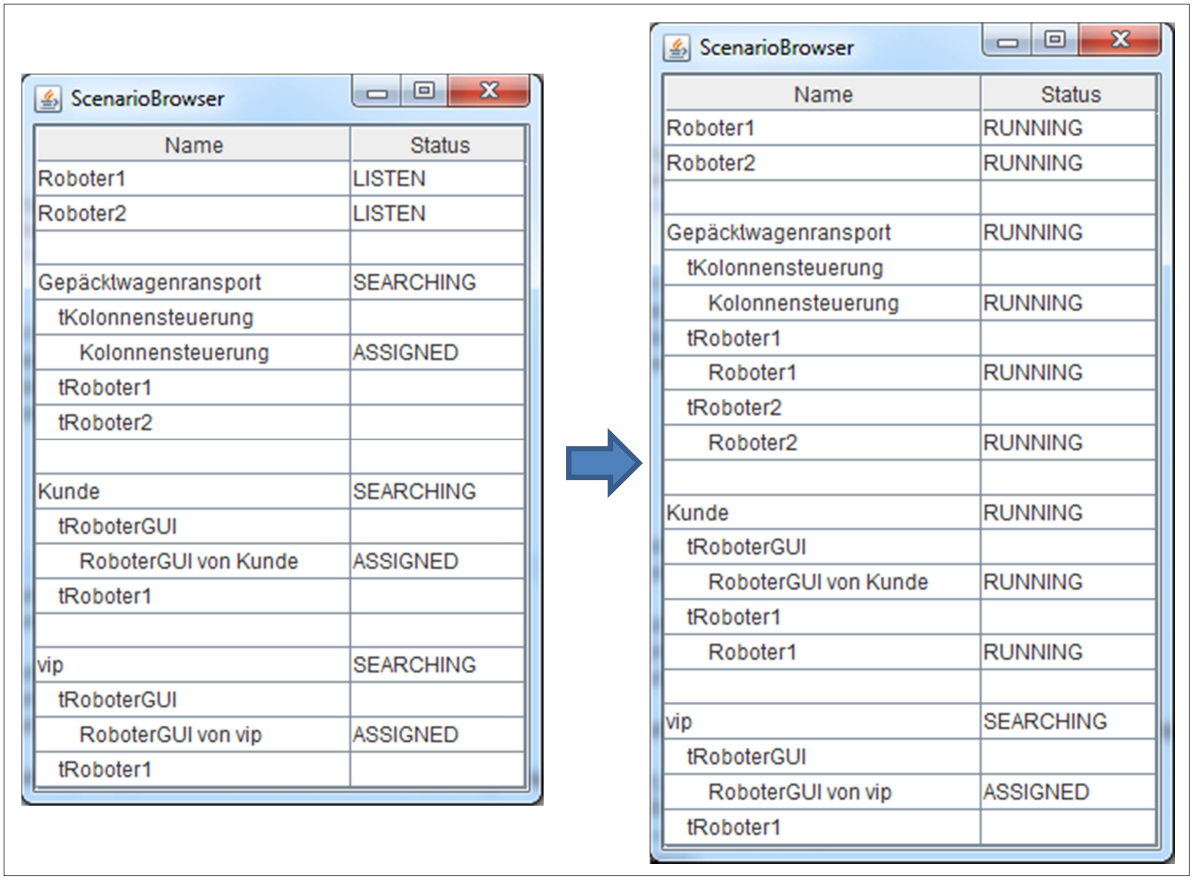

Abbildung 7-19: Testfall 4

Die Ausgangssystemkonfiguration in Abbildung 7-19 enthält, analog zur Ausgangssystemkonfiguration in Szenario 4, zwei freie Komponenten „Roboter1“ und „Roboter2“ und die drei Anwendungen „Kunde“, „Gepäckwagentransport“ und „vip“, siehe Abbildung 3-23. In der Zielsystemkonfiguration von Abbildung 7-19 sind die beiden Komponenten „Roboter1“ und „Roboter2“ der Anwendung „Gepäckwagentransport“ zugeordnet. Außerdem ist die Komponente „Roboter1“ noch der Anwendung „Kunde“ zugeordnet. Bis auf die Anwendung „vip“, welche sich noch im Zustand SEARCHING befindet, sind alle genannten Komponenten und Anwendungen im Zustand RUNNING. Das entspricht der Zielsystemkonfiguration von Szenario 4 des Anwendungsbeispiels, siehe Abbildung 3-23, womit das implementierte Verfahren auch alle Anforderungen des 4. Szenarios erfüllt.

7.6.5 Testfall 5

Im fünften Testfall wird Szenario 5 des Anwendungsbeispiels geprüft. Die Ausgangssystemkonfiguration und die erwartete Zielsystemkonfiguration sind in Abbildung 3-21 zu sehen.

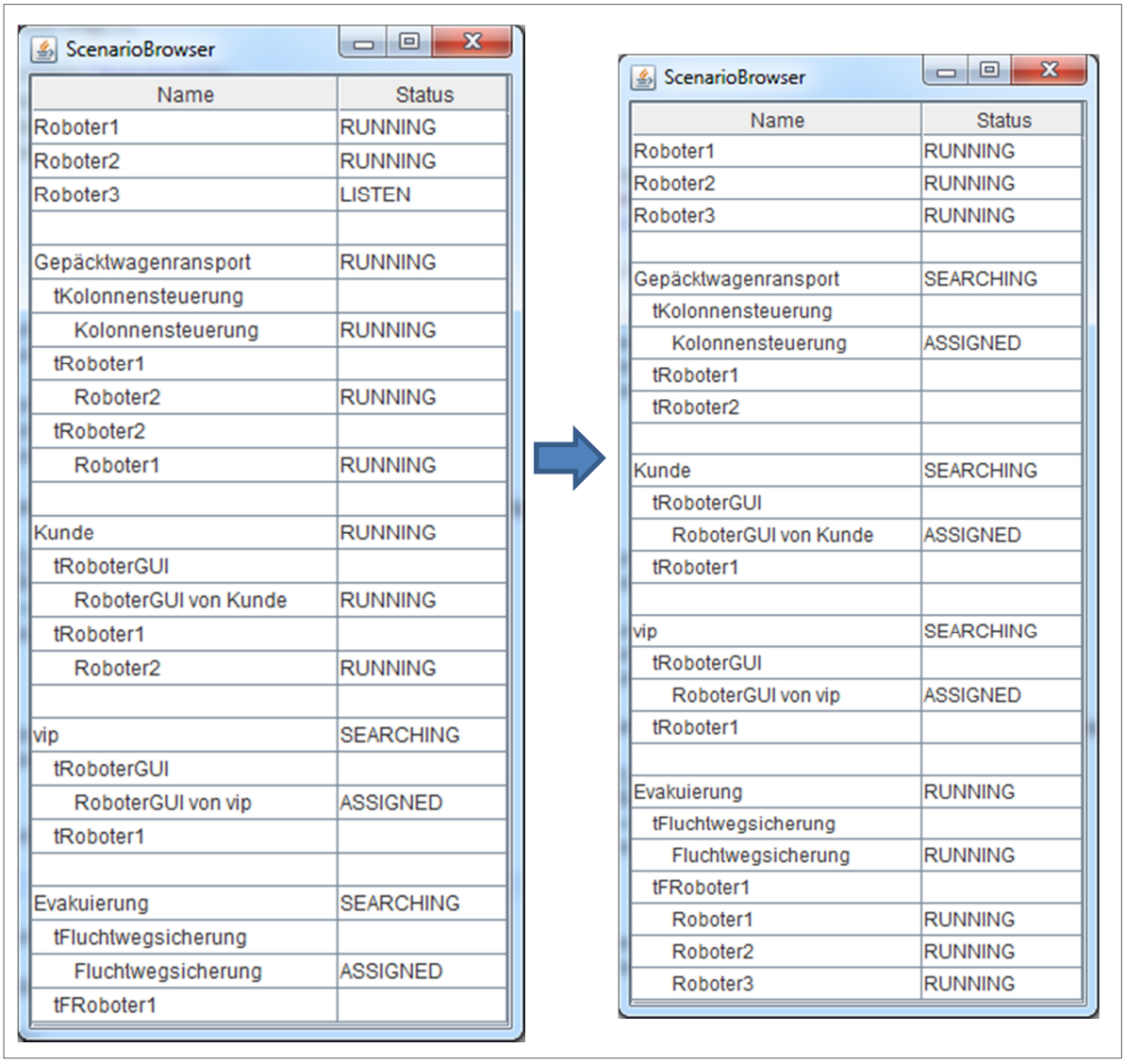

Abbildung 7-20: Testfall 5

Die Ausgangssystemkonfiguration in Abbildung 7-20 besitzt den gleichen Aufbau wie die Ausgangssystemkonfiguration des 5. Szenarios, siehe Abbildung 3-24. Sie besteht aus der Zielsystemkonfiguration des vierten Szenarios und zusätzlich aus einer Komponente „Roboter3“ und einer Anwendung „Evakuierung“. In der Zielsystemkonfiguration in Abbildung 7-20 wurden die Komponenten „Roboter1“, „Roboter2“ und „Roboter3“ der Anwendung „Evakuierung“ zugeordnet. Diese Anwendung und die drei genannten Komponenten befinden sich im Zustand RUNNING. Alle übrigen Anwendungen befinden sich im Zustand SEARCHING.

Die Zielsystemkonfiguration aus Abbildung 7-20 entspricht der Zielsystemkonfiguration aus Abbildung 3-24 des 5. Szenarios des Anwendungsbeispiels. Das implementierte Verfahren erfüllt demnach auch alle Anforderungen des letzten Szenarios des Anwendungsbeispiels.

7.6.6 Testfall 6

Der sechste Testfall soll aufzeigen, dass mit dem Verfahren auch komplexere Ausgangssystemkonfigurationen zu optimalen Zielsystemkonfigurationen verschaltet werden können.

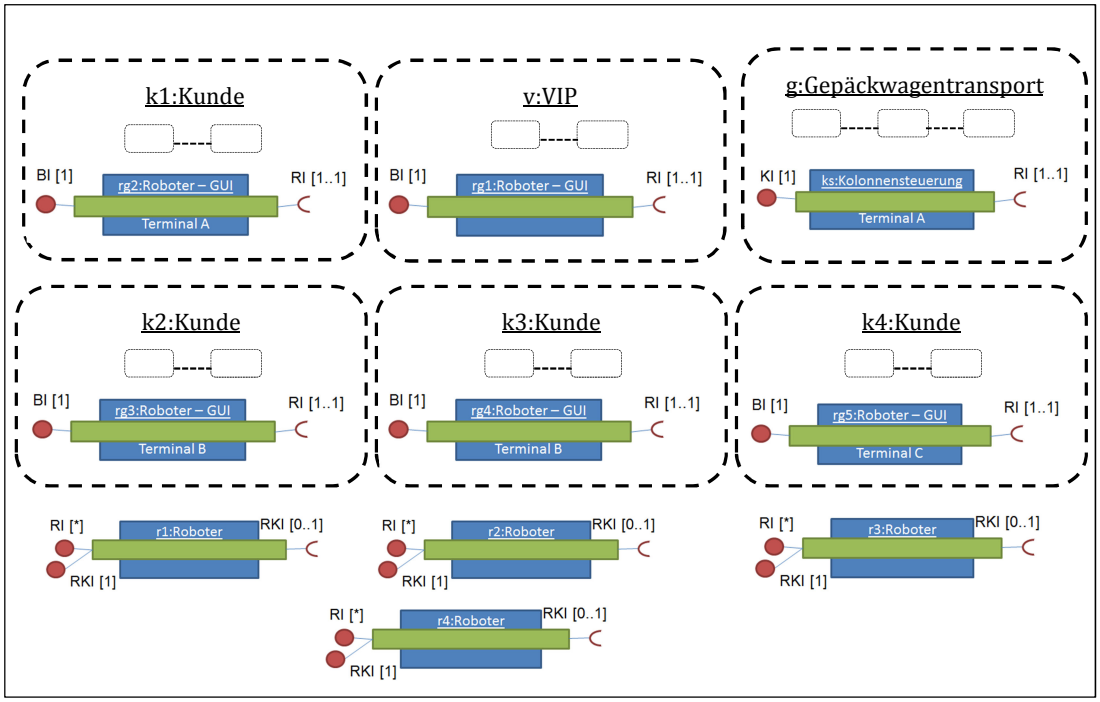

Abbildung 7-21: Komplexere Ausgangssystemkonfiguration unter Verwendung des Komponentenmodells

Das erste Zusatzszenario besteht, wie in Abbildung 7-21 dargestellt, aus vier Anwendungen vom Typ „Kunde“, einer Anwendung vom Typ „Gepäckwagentransport“, einer Anwendung vom Typ „VIP“. Außerdem enthält das Szenario vier freie Komponenten vom Typ „Roboter“. Die Gepäckwagentransportanwendung und eine Kundenanwendung besitzen beide das gleiche Ziel „Terminal A“. Zwei weitere Kundenanwendungen besitzen das Ziel „Terminal B“ und die letzte im System vorhandene Kundenanwendung besitzt das Ziel „Terminal C“. Nach der Zielfunktion aus Kapitel 4 ist die optimale Systemkonfiguration erreicht, wenn sich folgende Konfigurationen bilden. Die beiden Anwendungen „Gepäckwagentransport“ und „Kunde“, welche beide das gleiche Ziel besitzen, erhalten insgesamt zwei Roboterkomponenten. Die dritte Roboterkomponente wird den beiden Kundenanwendungen mit dem Ziel "Terminal B“ zugeordnet und die VIP-Anwendung erhält die letzte freie Roboterkomponente. Mit dieser Zuordnung kann der in dieser Zusammenstellung höchste Wert der Bewertungsfunktion erreicht werden.

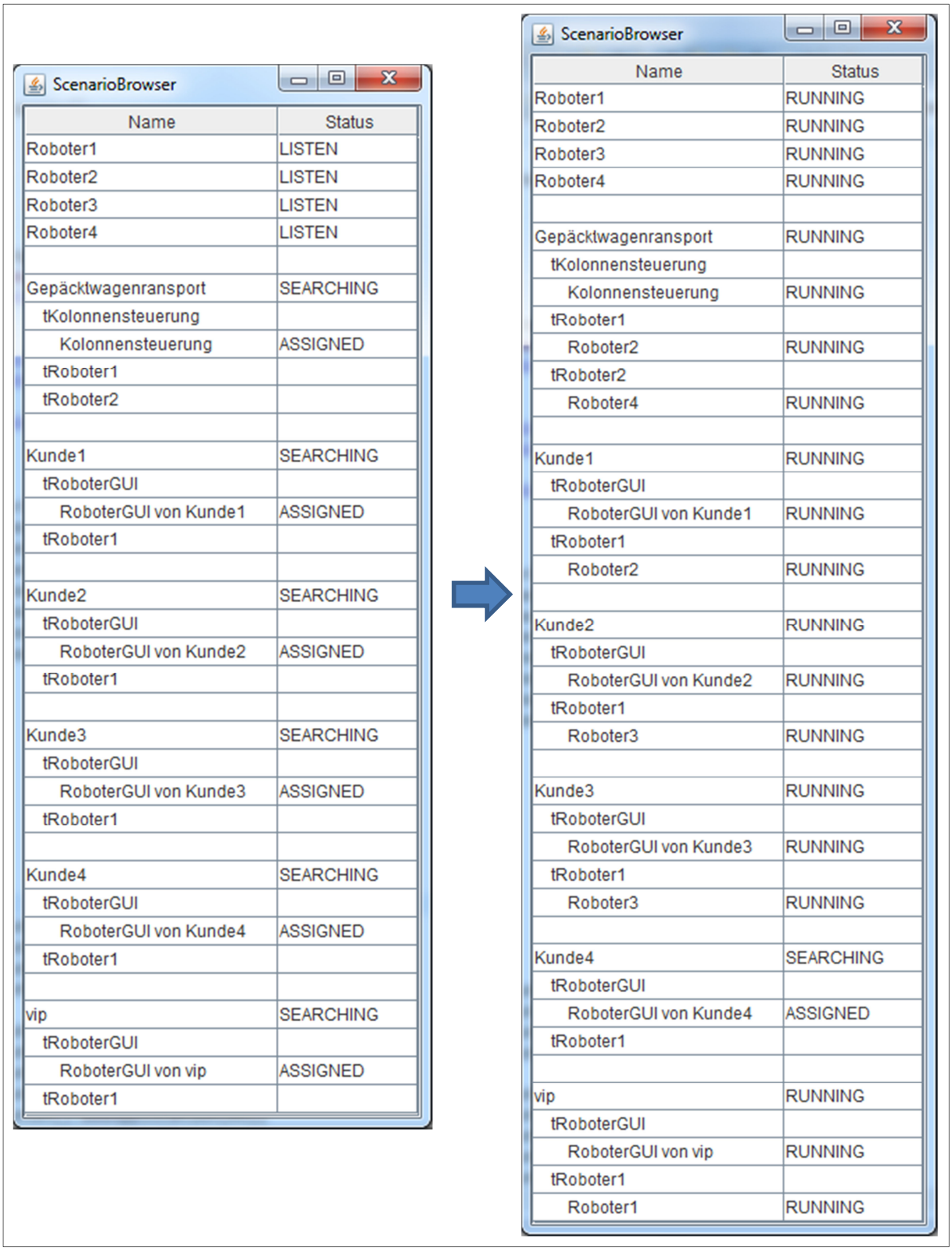

Abbildung 7-22: Testfall 6

In Abbildung 7-22 wird die Ausgangssystemkonfiguration des sechsten Testfalls dargestellt, welche der Ausgangssystemkonfiguration des 1. Zusatzszenarios entspricht. In der Zielsystemkonfiguration befinden sich alle Anwendungen bis auf Anwendung „Kunde 4“, welche als Ziel „Terminal C“ besitzt, im Zustand RUNNING.

Damit entspricht die Zielsystemkonfiguration der erwarteten Zielsystemkonfiguration des 1. Zusatzszenarios. Das implementierte Verfahren kann daher auch komplexere Ausgangssystemkonfigurationen korrekt verschalten.

7.6.7 Testfall 7

Der letzte Testfall soll mit einem weiteren Zusatzszenario zeigen, dass mit dem implementierten Verfahren ebenfalls eine optimale Zielsystemkonfiguration gewählt wird, wenn mehrere optimale Zielsystemkonfigurationen im System existieren.

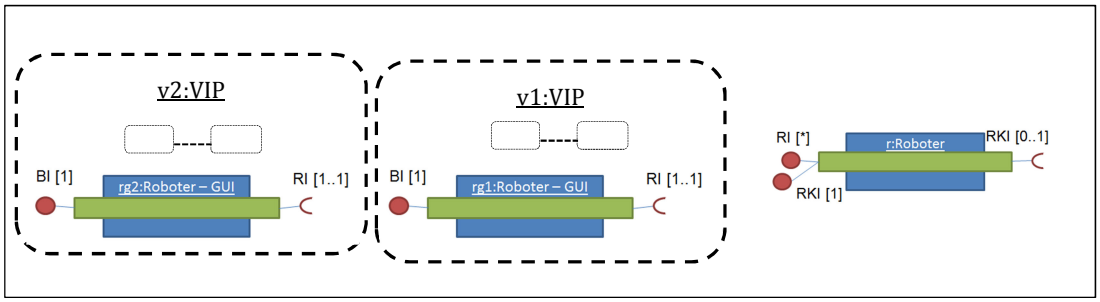

Abbildung 7-23: Ausgangssystemkonfiguration unter Verwendung des Komponentenmodells

In Abbildung 7-23 werden als Ausgangssystemkonfiguration des zweiten Zusatzszenarios zwei VIP-Anwendungen und eine freie Roboterkomponente dargestellt. Die Roboterkomponente kann nur an eine der beiden VIP-Anwendungen gebunden werden. Allerdings liefert die Bewertungsfunktion bei beiden Varianten exakt den gleichen Wert. Das Verfahren aus Kapitel 5 muss sich demnach für eine optimale Zielsystemkonfiguration entscheiden können.

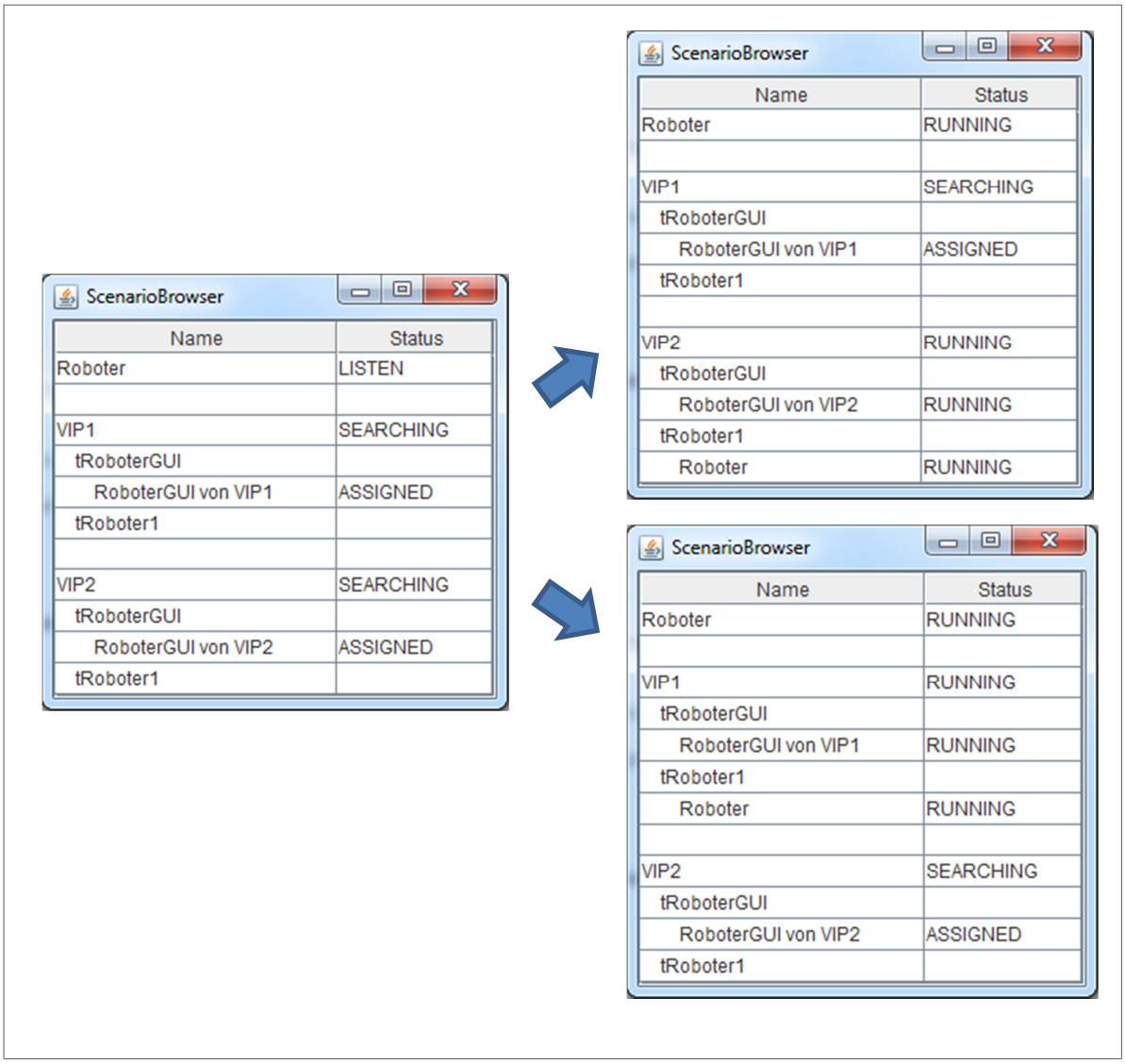

Abbildung 7-24: Testfall 7

Die Ausgangssystemkonfiguration des siebten Testfalls wird in Abbildung 7-24 dargestellt. Sie entspricht der Ausgangssystemkonfiguration des zweiten Zusatzszenarios. Außerdem werden in Abbildung 7-24 zwei Zielsystemkonfigurationen, eine für jede optimale Zielsystemkonfiguration des zweiten Zusatzszenarios, dargestellt. Welche Zielsystemkonfiguration das implementierte Verfahren erstellt, ist dem Zufall überlassen, aber pro Anwendung des Verfahrens auf die Ausgangsystemkonfiguration wird eine der beiden Zielsystemkonfigurationen ausgewählt und wechselt nicht zwischen beiden Zielsystemkonfigurationen. Somit erfüllt das implementierte Verfahren auch die Anforderungen des 2. Zusatzszenarios.

7.7 Zusammenfassung

In diesem Kapitel wurden alle Szenarien des Anwendungsbeispiels aus Kapitel 3 und zwei zusätzliche Szenarien mit einer Implementierung des Verfahrens aus Kapitel 5 ausgeführt. Alle Testfälle liefern exakt die erwarteten Zielsystemkonfigurationen der jeweiligen Szenarien. Daher ist das Verfahren aus Kapitel 5 eine Lösung für die in Kapitel 3 erarbeitete Problemstellung.

8 Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst. Anschließend wird ein Ausblick auf mögliche Erweiterungen und Verbesserungen des Lösungsverfahrens gegeben.

8.1 Zusammenfassung

In dieser Arbeit wurde ein Ansatz vorgestellt, der eine Menge von konkurrierenden Anwendungen dezentral zu einer optimalen Systemkonfiguration verschalten kann. Zur Erläuterung der Problemstellung wurde ein Anwendungsbeispiel aus dem Bereich eines Smart-Airport präsentiert.

In dem Anwendungsbeispiel existiert eine begrenzte Menge an Gepäckwagenrobotern, welche in vier Anwendungen unterschiedliche Aufgaben zu verrichten haben. Die Gepäckwagenroboter wurden nun in 5 verschiedenen Szenarien optimal verteilt. Dabei wurden die Bedingungen, welche die Anwendungen untereinander besitzen, berücksichtigt.

Aus diesen Szenarien wurde eine Bewertungsfunktion abgeleitet, über welche anschließend die optimale Verteilung aller Szenarien berechnet werden konnte. Anhand dieser Bewertungsfunktion konnte nun ein dezentrales Verfahren zur Erstellung der optimalen Systemkonfiguration entwickelt werden.

Dazu wurden spezielle Anforderungen aus dem Anwendungsbeispiel herausgearbeitet, die ein mögliches Verfahren beim Verschalten der Anwendungen in einzelnen Szenarien erfüllen muss. Anschließend wurde ein Verfahren vorgestellt, welches jeweils eine Menge von Instanzen der vier Anwendungen unter Berücksichtigung aller ermittelten Anforderungen optimal verschalten kann.

Das vorgestellte Verfahren wurde in einem Prototyp in der Programmiersprache Java zusammen mit den Szenarien des Anwendungsbeispiels implementiert. Anhand der Szenarien wurde das implementierte Verfahren und damit das vorgestellte Verfahren evaluiert. Zwei weitere Szenarien zeigten außerdem, dass das vorgestellte Verfahren nicht nur auf das Anwendungsbeispiel beschränkt ist.

Zur Bearbeitung der Problemstellung wurden vier Forschungsfragen aufgestellt und untersucht. In der ersten Forschungsfrage wurde eine Definition für eine optimale Systemkonfiguration eines Systems gesucht. Dafür musste zuerst einmal das System an sich definiert werden. Anschließend mussten die verschiedenen Systemkonfigurationen beschrieben werden, welche sich in dem System einstellen können. Danach wurde nach einer Bewertungsfunktion gesucht, welche eine Systemkonfiguration bewerten kann, um eine Sortierung der Systemkonfigurationen zu ermöglichen. Schließlich konnte eine Zielfunktion definiert werden, welche über die Bewertungsfunktion die optimale Systemkonfiguration ermitteln kann. Dies wurde in Kapitel 4.1 ausführlich beschrieben. Daher wird diese Forschungsfrage als beantwortet angesehen.

In der zweiten Forschungsfrage wurde nach den Eigenschaften eines dezentralen Algorithmus gefragt, die das sichere Finden einer optimalen Systemkonfiguration gewährleisten können. Für das sichere Finden einer optimalen Systemkonfiguration wurden analytische Verfahren als die geeigneten Verfahren ausgewählt. Heuristische Verfahren zum Beispiel können gute Systemkonfigurationen finden, aber bei ihnen kann nicht garantiert werden, dass immer eine optimale Systemkonfiguration gefunden wird. Untersucht wurde die Forschungsfrage in Kapitel 4.2. Außer-

dem wurden zu den Anforderungen dieser Forschungsfrage Lösungsansätze genannt. In Kapitel 5 wurde das entwickelte Verfahren dann ausführlich beschrieben. Da dieses Verfahren alle möglichen Anwendungskonfigurationen und Anwendungsdomänenkonfigurationen als Teil einer Systemkonfiguration ermittelt, werden auch alle Teile der optimalen Systemkonfiguration erzeugt. Diese Teile werden dann im Verfahren ermittelt und zu der optimalen Systemkonfiguration zusammengebaut. Daher liefert dieses Verfahren immer die optimale Systemkonfiguration. Somit wurde auch diese Frage hinreichend beantwortet.

In der dritten Forschungsfrage ging es um den Wunsch nach Stabilität des Verfahrens und des zu erreichenden Optimums. Bei mehreren optimalen Systemkonfigurationen darf das Verfahren nicht zwischen diesen Konfigurationen pendeln, sondern muss sich für eine Konfiguration entscheiden. Außerdem soll das Verfahren nicht sofort eine bestehende Konfiguration löschen, wenn durch das Hinzukommen einer neuen Anwendung eine bessere Konfiguration möglich ist. Ansonsten kann eine z. B. instabile Anwendung, welche das System immer wieder verlässt und betritt, erreichen, dass das System nicht mehr lauffähig ist. In Kapitel 4.3 wurde diese Frage untersucht. Darin wurde festgelegt, dass eine bestehende Verschaltung bestehen bleibt, wenn sich das System nur durch das Hinzukommen von Komponenten oder Anwendungen gleicher Priorität, wie die der verschalteten Anwendungen, verändert. Wenn hingegen eine Anwendung höherer Priorität das System betritt, soll eine neue Konfiguration gesucht werden können. Die bestehenden Verschaltungen werden aber nur gelöst, nachdem eine neue Konfiguration gefunden wurde, was in diesem System eine hohe Stabilität gewährleistet. Damit ist auch diese Frage hinreichend beantwortet worden.

In der letzten Forschungsfrage wurde nach einer Möglichkeit gesucht, die daraus entwickelte dezentrale Middleware zu evaluieren. Darauf wurde in Kapitel 4.4 eingegangen. Für die Evaluierung wurden Testfälle aus den Szenarien des Anwendungsbeispiels aus Kapitel 3 erstellt. Anschließend wurde in Kapitel 6 ein Prototyp vorgestellt und in Kapitel 7 wurden die Testfälle anhand des Prototyps durchgeführt. Da die Forschungsfragen und die Problemstellung aus diesem Anwendungsbeispiel abgeleitet wurden, konnte somit gezeigt werden, dass die entwickelte Middleware die nötigen Anforderungen erfüllt. Mit zwei zusätzlichen Testfällen konnte außerdem gezeigt werden, dass die Middleware nicht nur auf das Anwendungsbeispiel beschränkt ist, womit auch die letzte Forschungsfrage beantwortet wurde.

8.2 Diskussion und Ausblick

Wie im letzten Abschnitt beschrieben, liefert das vorgestellte Verfahren eine gute Lösung für die Problemstellung. Allerdings besitzt es auch einige Schwächen, die Potenzial für weitere Forschungen liefern. Eine Schwäche besteht in der Verwendung eines analytischen Verfahrens. Damit wird zwar das sichere Erreichen einer optimalen Systemkonfiguration garantiert. Da dieser Ansatz aber NP-vollständig ist, steigen der Rechenaufwand und Speicherverbrauch exponentiell mit der Anzahl der Anwendungen und Komponenten an. Daher ist das beschriebene Verfahren in der Praxis bei einer größeren Domäne nicht anwendbar. Schon bei Testfall 6, in dem vier Gepäckwagenroboter auf sechs Anwendungen verteilt werden, existierten so viele Kombinationsmöglichkeiten, dass der Rechner einige Minuten für die relativ einfache optimale Zuordnung benötigte. Ein Austausch der Berechnung für alle möglichen Anwendungskonfigurationen einer Anwendung und das Kombinieren aller Anwendungskonfigurationen in einer Clique durch heuristische Verfahren können hier mögliche Lösungen sein. Damit kann zwar nicht mehr garantiert werden, dass die optimale Systemkonfiguration gefunden wird, aber evtl. wird dies ja auch nicht überall gefordert. Dafür kann mit akzeptablem Ressourcenaufwand eine ausreichend gute Sys-

temkonfiguration gefunden werden. Für die Frage nach den passenden heuristischen Ansätzen ist allerdings weitere Forschungsarbeit notwendig.

Des Weiteren wurden in dieser Arbeit nur Komponenten mit genau einer Konfiguration verwendet. Daher war der Wert aus der Bewertungsfunktion einer lauffähigen Anwendung meist konstant. Bei der Evakuierungsanwendung war der Wert von der Anzahl der Roboterkomponenten abhängig. Aber unterschiedliche Konfigurationen einer Komponente wurden dabei nicht berücksichtigt. Das heißt, Anwendungen mit Komponenten in einer hohen Konfiguration besitzen momentan den gleichen Wert wie die gleichen Anwendungen mit Komponenten in einer niedrigeren Konfiguration. Da die erstgenannten Anwendungen aber besser, effektiver oder mit mehr Funktionen genutzt werden können, sollten sie vor den zweitgenannten Anwendungen verschaltet werden können. Allerdings stellen sich hier weitere Forschungsfragen, z. B. ob eine Anwendung in einer besseren Konfiguration arbeiten darf, wenn dabei eine andere Anwendung gleicher Priorität dafür keine Anwendungskonfiguration erstellen kann. Dafür ist evtl. sogar eine ganz neue Bewertungsfunktion notwendig.

Zusammenfassend wurde in dieser Arbeit ein Verfahren erforscht, um eine Menge von Komponenten dezentral zu einer optimalen Menge von Anwendungen zu verschalten. Das vorgestellte Verfahren erfüllt alle dabei aufgestellten Anforderungen. Es ermittelt immer die optimale Verschaltung. Aufgrund des exponentiellen Rechenaufwands kann das Verfahren allerdings nur in kleineren Anwendungsdomänen mit wenigen Anwendungsinstanzen und Komponenten eingesetzt werden.

Abbildungsverzeichnis

Literaturverzeichnis

Arcaini, P.; Riccobene, E.; Scandurra, P. (2015): Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation. In: 10th International Symposium on 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), S. 13–23.

Backschat, Martin; Rücker, Bernd (2007): Enterprise JavaBeans 3.0. In: *Grundlagen-Konzepte-Praxis, Spektrum-Verlag*.

Becker, C.; Handte, M.; Schiele, G.; Rothermel, K. (2004): PCOM - a component system for pervasive computing. In: Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. Proceedings of the. Orlando, FL, USA, S. 67–76.

Bernstein, Philip A. (1996): Middleware: A Model for Distributed System Services. In: *Commun. ACM* 39 (2), S. 86–98. DOI: 10.1145/230798.230809.

Chavez, Anthony; Moukas, Alexandros; Maes, Pattie (1997): Challenger: A Multi-agent System for Distributed Resource Allocation. In: Proceedings of the First International Conference on Autonomous Agents. New York, NY, USA: ACM (AGENTS '97), S. 323–331.

Costa, P.; Coulson, G.; Mascolo, C.; Picco, G. P.; Zachariadis, S. (2005): The RUNES middleware: a reconfigurable component-based approach to networked embedded systems. In: Personal, Indoor and Mobile Radio Communications, 2005. PIMRC 2005. IEEE 16th International Symposium on, Bd. 2, S. 806-810 Vol. 2.

Dennisen, Sophie L.; Müller, Jörg P. (2015): Agent-Based Voting Architecture for Traffic Applications. Multiagent System Technologies : 13th German Conference, MATES 2015, Cottbus, Germany, September 28 - 30, 2015, Revised Selected Papers. In: P. Jörg Müller, Wolf Ketter, Gal Kaminka, Gerd Wagner und Nils Bulling (Hg.). Cham: Springer International Publishing, S. 200–217.

Di Nitto, Elisabetta; Ghezzi, Carlo; Metzger, Andreas; Papazoglou, Mike; Pohl, Klaus (2008): A journey to highly dynamic, self-adaptive service-based applications. In: *Automated Software Engineering* 15 (3-4), S. 313–341. DOI: 10.1007/s10515-008-0032-x.

Dorigo, Marco; Stützle, Thomas (2009): Ant colony optimization: overview and recent advances. In: *Techreport, IRIDIA, Universite Libre de Bruxelles*.

Dowling, Jim (2004): The decentralised coordination of self-adaptive components for autonomic distributed systems. Citeseer.

Downing, Troy (1998): Java RMI. Remote method invocation. Foster City, CA: IDG Books Worldwide.

Gharbi, Mahbouba; Koschel, Arne; Rausch, Andreas; Starke, Gernot (2015): Basiswissen für Softwarearchitekten. Aus- und Weiterbildung nach iSAQB-Standard zum Certified Professional for Software Architecture - Foundation Level. 2. Auflage. Heidelberg: dpunkt.verlag.

Goerzen, C.; Kong, Z.; Mettler, B. (2009): A Survey of Motion Planning Algorithms from the Perspective of Autonomous UAV Guidance. In: *Journal of Intelligent and Robotic Systems* 57 (1), S. 65–100. DOI: 10.1007/s10846-009-9383-1.

Heineman, George T.; Councill, William T. (2001): Component-based software engineering. Putting the pieces together. Boston: Addison-Wesley.

Herold, Sebastian; Klus, Holger; Niebuhr, Dirk; Rausch, Andreas (2008): Engineering of IT ecosystems. In: Kevin Sullivan und Rick Kazman (Hg.): the 2nd international workshop. Leipzig, Germany, S. 49–52.

Huebscher, Markus C.; McCann, Julie A. (2008): A Survey of Autonomic Computing&Mdash;Degrees, Models, and Applications. In: *ACM Comput. Surv.* 40 (3), S. 7:1. DOI: 10.1145/1380584.1380585.

Jensen, Andreas Schmidt; Dignum, Virginia; Villadsen, Jørgen (2016): A framework for organization-aware agents. In: *Autonomous Agents and Multi-Agent Systems*, S. 1–36. DOI: 10.1007/s10458-015-9324-2.

Jifeng, He; Li, Xiaoshan; Liu, Zhiming (2005): Component-Based Software Engineering. In: Dang van Hung und Martin Wirsing (Hg.): Theoretical Aspects of Computing – ICTAC 2005, Bd. 3722: Springer Berlin Heidelberg (Lecture Notes in Computer Science), S. 70–95.

Keller, Marco; Pütz, Stefan; Siml, Jan (2012): Internet der Dinge. In: *A. Mehler-Bicher & L. Steiger Trends in der IT*, S. 118–122.

Klus, Holger (2013): Anwendungsarchitektur-konforme Konfiguration selbstorganisierender Softwaresysteme: Dr. Hut Verlag (SSE-Dissertation).

Klus, Holger; Niebuhr, Dirk; Rausch, Andreas (2007): A component model for dynamic adaptive systems. In: International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting. New York, NY, USA: ACM (ESSPE '07), S. 21–28.

Leclercq, Matthieu; Quéma, Vivien; Stefani, Jean-Bernard (2005): DREAM: A Component Framework for Constructing Resource-Aware, Configurable Middleware. In: *IEEE Distributed Systems Online* 6 (9), S. 1. DOI: 10.1109/MDSO.2005.47.

Lemos, Rogério de; Giese, Holger; Müller, HausiA.; Shaw, Mary; Andersson, Jesper; Litoiu, Marin et al. (2013): Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: Rogério de Lemos, Holger Giese, HausiA. Müller und Mary Shaw (Hg.): Software Engineering for Self-Adaptive Systems II, Bd. 7475: Springer Berlin Heidelberg (Lecture Notes in Computer Science), S. 1–32.

Liu, Hua; Parashar, M. (2006): Accord: a programming framework for autonomic applications. In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 36 (3), S. 341–352. DOI: 10.1109/TSMCC.2006.871577.

Millard-Ball, Adam; Weinberger, Rachel R.; Hampshire, Robert C. (2014): Is the curb 80% full or 20% empty? Assessing the impacts of San Francisco's parking pricing experiment. In: *Transportation Research Part A: Policy and Practice* 63, S. 76–92. DOI: 10.1016/j.tra.2014.02.016.

Mogelmose, A.; Trived, M. M.i; Moeslund, T. B. (2012): Vision-Based Traffic Sign Detection and Analysis for Intelligent Driver Assistance Systems: Perspectives and Survey. In: *IEEE Transactions on Intelligent Transportation Systems* 13 (4), S. 1484–1497. DOI: 10.1109/TITS.2012.2209421.

Nami, Mohammad Reza; Sharifi, Mohsen (2007): A Survey of Autonomic Computing Systems. Intelligent Information Processing III: IFIP TC12 International Conference on Intelligent Information Processing (IIP 2006), September 20-23, Adelaide, Australia. In: Zhongzhi Shi, K. Shimohara und D. Feng (Hg.). Boston, MA: Springer US, S. 101–110.

Nash, John F.; others (1950): Equilibrium points in n-person games. In: *Proc. Nat. Acad. Sci. USA* 36 (1), S. 48–49.

Netzer, Arnon; Meisels, Amnon; Zivan, Roie (2015): Distributed envy minimization for resource allocation. In: *Autonomous Agents and Multi-Agent Systems*, S. 1–39. DOI: 10.1007/s10458-015-9291-7.

Neumann, John von (1928): Zur Theorie der Gesellschaftsspiele. In: *Mathematische Annalen* 100 (1), S. 295–320.

Neumann, John von; Morgenstern, Oskar (2007): Theory of games and economic behavior: Princeton university press.

Nuseibeh, B. (2001): Weaving together requirements and architectures. In: *Computer* 34 (3), S. 115–119. DOI: 10.1109/2.910904.

Oreizy, P.; Gorlick, M.M; Taylor, R.N; Heimhigner, D.; Johnson, G.; Medvidovic, N. et al. (1999): An architecture-based approach to self-adaptive software. In: *IEEE Intell. Syst.* 14 (3), S. 54–62. DOI: 10.1109/5254.769885.

Parashar, M.; Liu, H.; Li, Z.; Matossian, V.; Schmidt, C.; Zhang, G.; Hariri, S. (2006): AutoMate: Enabling Autonomic Applications on the Grid. In: *Cluster Comput* 9 (2), S. 161–174. DOI: 10.1007/s10586-006-7561-5.

Petty, Mikel D.; Weisel, Eric W.; Mielke, Roland R. (2003): Computational complexity of selecting components for composition. In: Proceedings of the Fall 2003 Simulation Interoperability Workshop. Citeseer, S. 14–19.

Russell, Stuart J.; Norvig, Peter (2012): Künstliche Intelligenz. Ein moderner Ansatz. 3., aktualisierte Aufl. München [u.a.]: Pearson (Pearson-Studium).

Rütschlin, Jochen; Sauter, Günter; Sellentin, Jürgen; Hergula, Klaudia; Mitschang, Bernhard (2001): Komponenten-Middleware der nächste Schritt zur Interoperabilität von IT-Systemen. In: Datenbanksysteme in Büro, Technik und Wissenschaft. Springer, S. 322–331.

Siegel, Jon (2000): Corba 3 fundamentals and programming. 2nd ed. New York: John Wiley & Sons.

Skog, I.; Handel, P. (2009): In-Car Positioning and Navigation Technologies #x2014;A Survey. In: *IEEE Transactions on Intelligent Transportation Systems* 10 (1), S. 4–21. DOI: 10.1109/TITS.2008.2011712.

Standish Group International, Inc (1999): CHAOS: A Recipe for Success.

Sykes, Daniel; Magee, Jeff; Kramer, Jeff (2011): FlashMob: Distributed Adaptive Self-assembly. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. New York, NY, USA: ACM (SEAMS '11), S. 100–109.

Szyperski, Clemens (2003): Component software. Beyond object-oriented programming. Reading: Addison-Wesley.

Turner, Mark; Budgen, David; Brereton, Pearl (2003): Turning software into a service. In: *Computer* 36 (10), S. 38–44.

Tuttlies, Verena; Schiele, Gregor; Becker, Christian (2007): COMITY - Conflict Avoidance in Pervasive Computing Environments. In: Robert Meersman, Zahir Tari und Pilar Herrero (Hg.): On

the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, Bd. 4806: Springer Berlin Heidelberg (Lecture Notes in Computer Science), S. 763–772.

Wang, Nanbor; Schmidt, Douglas C.; O'Ryan, Carlos (2001): Overview of the corba component model. In: Component-Based Software Engineering. Addison-Wesley Longman Publishing Co., Inc, S. 557–571.

Weishäupl, Thomas (2002): Component-based Software Engineering. Online verfügbar unter http://eprints.cs.univie.ac.at/1171/, zuletzt geprüft am 13.04.2016.

Wooldridge, Michael J. (2009): An introduction to multiagent systems. 2nd ed. Chichester, U.K.: John Wiley & Sons.