



Hochschule für Angewandte Wissenschaften Hamburg  
*Hamburg University of Applied Sciences*

# Masterarbeit

Sven Tennstedt

**Szenarien basierte Interpretationen eines Smart-Homes**

*Fakultät Technik und Informatik  
Studiendepartment Informatik*

*Faculty of Engineering and Computer Science  
Department of Computer Science*

Sven Tennstedt

**Szenarien basierte Interpretationen eines Smart-Homes**

Betreuender Prüfer: Prof. Dr. rer. nat. Kai von Luck  
Zweitgutachterin: Prof. Dr. Julia Padberg

Eingereicht am: 27. Oktober 2011

**Sven Tennstedt**

**Thema der Arbeit**

Szenarien basierte Interpretationen eines Smart-Homes

**Stichworte**

ubiquitäres Computing, Context Awareness, intelligente Wohnung, automatisches Planen, Verhaltens Interpretation, Companion Technology, Intervallalgebra

**Kurzzusammenfassung**

In dieser Arbeit wird ein System zur szenarienbasierten Interpretation der Lebensabläufe in einem Smart-Home entwickelt. Die bisherige Forschung konzentriert sich weitestgehend auf die technische Umsetzung der dynamischen Anforderungen, sowie die Reaktion auf bestimmte Situationen.

In dieser Arbeit wird ein Smart-Home dahingehend erweitert, dass anhand vorher erkannter Szenarien Aktionen ausgeführt werden, die den Bewohner in seinen Lebensabläufen vorausschauend unterstützen. Anstelle von Aktionsplänen wie in der klassischen Planung, wird eine Zielesemantik verwendet. Anhand der aktuellen Situation, innerhalb eines erkannten Szenarios, werden die Ziele der Bedürfnissen des Bewohners angepasst und von einer Agentengesellschaft erfüllt.

**Sven Tennstedt**

**Title of the paper**

Scenario Based Interpretation of a Smart-Home

**Keywords**

ubiquitous computing, context awareness, Smart-Home, automated planning, behaviour interpretation, companion technology, interval algebra

**Abstract**

Within this thesis a system will be developed that interprets the daily routine in a Smart-Home based on scenarios. The current research focuses on technical implementations of dynamic requirements as well as on reactions on certain situations. The thesis extends the Smart-Home to execute actions based on a previously identified scenario, to support the inhabitant anticipatorily on his/her daily routine. Instead of actionplans (used by classical planning) a goal semantic is used. Based on the current situation within the identified scenario goals are adjusted to the needs of the inhabitant. These goals are reached by an agent society.

# Inhaltsverzeichnis

<b>Tabellenverzeichnis</b>	<b>vii</b>
<b>Abbildungsverzeichnis</b>	<b>viii</b>
<b>1. Einführung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Zielsetzung . . . . .	2
1.3. Aufbau der Arbeit . . . . .	3
<b>2. Grundlagen</b>	<b>4</b>
2.1. Intelligente Wohnungen . . . . .	4
2.1.1. Ubiquitous Computing . . . . .	4
2.1.2. Kontext und Context-Awareness . . . . .	5
2.2. Stand der Forschung . . . . .	6
2.2.1. ubiHome . . . . .	7
2.2.2. Personal Home Server . . . . .	9
2.2.3. HomeLab . . . . .	10
2.2.4. CoFriend - Behaviour-Interpretation . . . . .	10
2.2.5. Living Place als Petrinetz . . . . .	12
2.3. Architektur des Living Places . . . . .	13
2.4. Automatisches Planen . . . . .	14
2.5. Zeit als Ressource . . . . .	17
2.5.1. Intervallalgebra . . . . .	17
2.5.2. Effizient lösbare Unterklassen von Allens Intervallalgebra . . . . .	19
2.5.3. Fazit zur Zeitalgebra . . . . .	20
<b>3. Analyse</b>	<b>21</b>
3.1. Zwei Userstories . . . . .	21
3.1.1. Userstory 1: Tagesbeginn . . . . .	22
3.1.2. Userstory 2: Besuch von Freunden . . . . .	22
3.1.3. Analyse der Userstories . . . . .	23
3.2. Charakterisierung der Ziele . . . . .	25
3.2.1. Herleitung der Ziele aus den Userstories . . . . .	26
3.2.2. Zieltypen . . . . .	27
3.2.3. Zeitrelation der Ziele untereinander . . . . .	28
3.2.4. Voraussetzung zum Erreichen eines Ziels . . . . .	30

3.2.5.	Definition der Zieldeklaration . . . . .	32
3.3.	Herleiten des Szenarios . . . . .	32
3.3.1.	Systematische Darstellung . . . . .	33
3.3.2.	Obligatorischer Pfad . . . . .	34
3.3.3.	Einsatz der Intervallalgebra . . . . .	37
3.4.	Zusammenfassung . . . . .	37
3.5.	Erkenntnisse der Analyse . . . . .	39
3.6.	Ermittelte Anforderungen an das Design . . . . .	40
<b>4.</b>	<b>Design und Realisierung</b> . . . . .	<b>43</b>
4.1.	Systembeschreibung . . . . .	43
4.1.1.	Sensordaten-Interpretation . . . . .	43
4.1.2.	Scenario-Reasoning . . . . .	44
4.1.3.	Behaviour-Prediction und System-Configuration . . . . .	45
4.2.	Architektur . . . . .	45
4.3.	Goal-Manager . . . . .	46
4.3.1.	Zustände von Szenarios und Zielen . . . . .	47
4.3.2.	Ziele finden mittels Graphensuche . . . . .	48
4.3.3.	Zeitmodell . . . . .	49
4.3.4.	Regeln für die Auswahl von Folgezielen . . . . .	50
4.4.	Goal-Agents . . . . .	51
4.5.	Koordination der Agenten . . . . .	51
4.5.1.	Agenten übernehmen und erfüllen Ziele . . . . .	52
4.5.2.	Propagieren der Start-/Endzeiten von Zielen . . . . .	53
4.6.	Technologie . . . . .	54
4.7.	Systemaufbau . . . . .	54
4.8.	Klassenmodell . . . . .	56
4.8.1.	Klasse Goal . . . . .	56
4.8.2.	Klasse GoalRelation . . . . .	57
4.8.3.	Klasse Scenario . . . . .	57
4.8.4.	Klassen GoalManagementAgent und GoalManager . . . . .	58
4.9.	Kommunikation . . . . .	59
4.9.1.	Szenario-Nachrichten . . . . .	60
4.9.2.	Goal-Management-Nachrichten . . . . .	61
4.10.	Tests und Ergebnisse der Simulation . . . . .	63
4.10.1.	GoalManager . . . . .	63
4.10.2.	Ergebnisse der Simulation . . . . .	67
4.11.	Erkenntnisse aus Realisierung und Tests . . . . .	72
4.12.	Evaluation des Designs und der Realisierung . . . . .	72
<b>5.</b>	<b>Zusammenfassung und Ausblick</b> . . . . .	<b>74</b>
5.1.	Zusammenfassung . . . . .	74
5.2.	Ausblick . . . . .	75

<b>Literaturverzeichnis</b>	<b>76</b>
<b>A. Inhalt der CD</b>	<b>80</b>
<b>B. Scala Schnellübersicht</b>	<b>81</b>
B.1. Deklaration von Klassen . . . . .	81
B.2. Singletons . . . . .	82
B.3. Generics . . . . .	82
B.4. var/val: Variablen und Werte . . . . .	82
B.5. Functions . . . . .	82
B.6. Partial Functions . . . . .	83
<b>C. Quellcode</b>	<b>85</b>
C.1. Goal . . . . .	85
C.2. GoalTypes . . . . .	87
C.3. GoalRelation . . . . .	87
C.4. Scenario . . . . .	88
C.5. GoalManagerGraph . . . . .	93
<b>D. Tests und Versuchsaufbau</b>	<b>95</b>
D.1. Unittests für den GoalManager . . . . .	95
D.1.1. Unittests Userstory 1 . . . . .	95
D.1.2. Unittests Userstory 2 . . . . .	97
D.2. GoalAgent Beispiel . . . . .	98
D.3. Console-Log . . . . .	99
D.3.1. : Versuch 1 . . . . .	99
D.3.2. : Versuch 2 . . . . .	100
<b>E. ActiveMQ Actors</b>	<b>102</b>
E.1. ActiveMQ Actors UML . . . . .	102
E.2. ActiveMQ Actors Quellcode . . . . .	103
<b>F. Korrespondenz</b>	<b>106</b>
F.1. Email Mathieu Vallée . . . . .	106

# Tabellenverzeichnis

2.1.	Intervall-Relationen und ihre Endpunkt-Relationen (Allen, 1983, S. 8, Figure 1). . .	19
4.1.	Transitivitätstabelle für die Intervallklasse $G$ . . . . .	50
4.2.	Planen von Zielen mit Abstand $\geq 1$ zu Knoten A. Markierung bezogen auf transitive Relation von A zu C. . . . .	50
4.3.	Planen von Zielen mit Abstand $\geq 1$ von Knoten B. Markierung bezogen auf transitive Relation von B zu C. . . . .	51

# Abbildungsverzeichnis

2.1.	Abstraktionsebenen der Kontextverarbeitung zur Kategorisierung der Smart-Home Projekte . . . . .	7
2.2.	Architektur-Übersicht des ubiHome Projekts (Ha u. a., 2007, Fig. 2) . . . . .	8
2.3.	Architektur-Übersicht des Personal Home Servers (Nakajima und Satoh, 2006) . . . . .	9
2.4.	Upper model aus (Bohlken u. a., 2011) . . . . .	12
2.5.	Architekturskizze des Living Places aus Ellenberg u. a. (2011) . . . . .	13
2.6.	Architektur des Living Places . . . . .	14
2.7.	Planning als State-Transition-System betrachtet . . . . .	15
2.8.	Konzeptskizze des automatischen Planens . . . . .	16
2.9.	Beziehungen von Zeitintervallen aus Allen (1983). Die Relationen $<$ und $>$ werden in der vorliegenden Arbeit mit $b$ ( <i>before</i> ) und $a$ ( <i>after</i> ) bezeichnet. . . . .	18
2.10.	Graph für die Konvexitätsbedingung in der Intervallalgebra . . . . .	19
3.1.	Zieltypen unterschieden nach Zeitpunkt der Zielerfüllung. . . . .	27
3.2.	Grafische Darstellung der Intervallrelation Einkaufen zu Öffnungszeiten . . . . .	29
3.3.	Erste Skizze des Szenarios für Userstory 1 von Seite 22. . . . .	33
3.4.	Erste Skizze des Szenario-Skripts für Userstory 2 von Seite 22. . . . .	34
3.5.	obligatorische und optionale Ziele von Userstory 1 . . . . .	35
3.6.	obligatorische und optionale Ziele von Userstory 2 . . . . .	36
3.7.	Ziele von Userstory 1 mittels der Intervallalgebra in Beziehung gesetzt. . . . .	38
3.8.	Ziele von Userstors 2 mittels der Intervallalgebra in Beziehung gesetzt. . . . .	38
3.9.	abstraktes Vorgehensmodell . . . . .	41
4.1.	Schematische Darstellung der Funktionseinheiten zur Interpretation von Sensorinformationen und der Identifikation und Interpretation von Szenarios, sowie das Erfüllen von Zielen. . . . .	44
4.2.	Die Funktionalitäten der Szenario-Identifikation und -Interpretation werden von den beiden Softwaremodulen Reasoner und Goal-Manager übernommen. . . . .	45
4.3.	Architektur des Living Places . . . . .	46
4.4.	Zustände eines Szenarios . . . . .	47
4.5.	Zustände eines Zieles . . . . .	47
4.6.	Szenario-Graph des Goal-Managers. . . . .	48
4.7.	Propagieren der Start- und Endzeit zwischen zwei voneinander abhängigen Zielen. . . . .	53
4.8.	Skizze der Komponenten . . . . .	55
4.9.	UML-Modell von Zielen . . . . .	56
4.10.	UML-Modell von Zielen und Relationen . . . . .	57



## *Abbildungsverzeichnis*

---

4.11. UML-Modell von Szenarios. . . . .	58
4.12. Versuchsaufbau Tagesbeginn-Szenario . . . . .	68
4.13. Versuchsaufbau Besuch von Freunden-Szenario . . . . .	70

# 1. Einführung

Mobile intelligente Geräte wie Handys, Tablet-Computer oder Notebooks gehören inzwischen zum Alltag. In den Wohnungen befinden sich in der Regel mehr oder weniger intelligente Geräte wie Musikanlage, Fernseher, Videorekorder, Kühlschrank, Kaffeemaschine usw.. Durch Entwicklungen neuartiger Handy- und Tabletbetriebsysteme wird die Bedienung mobiler Geräte immer einfacher. Alle diese Geräte nutzt der Mensch ganz selbstverständlich. Doch arbeiten diese Geräte nicht oder nur teilweise zusammen. Optimal funktioniert dies oft nur mit einer homogenen Diensteanbieterlandschaft. Das hat eine Fragmentierung der Informationsbeschaffung und eine Fragmentierung der Bedienung zur Folge. Für das Gelingen der Zusammenarbeit der einzelnen Geräte bzw. vielmehr der einzelnen Applikationen ist es oft notwendig für alles die Dienste ausschließlich eines Anbieters zu nutzen.

In [Weiser \(1991\)](#) wurde 1991 die Vision des Ubiquitous Computing geschaffen. Eine Vision, dass Computer irgendwann als eigenständige technische Geräte immer weiter in den Hintergrund treten und stattdessen viele kleine Geräte überall und jederzeit die Informationen zur Verfügung stellen, die wir benötigen. Die Vision ging noch darüber hinaus. Die Geräte sollen den Menschen viel mehr in seinem Alltag aktiv unterstützen, in dem sie Informationen automatisch aufbereiten und anbieten. Zudem sollen intelligente Haushaltsgeräte und Roboter Dienste anbieten und verrichten. Diese Vision wird im Forschungsbereich des Smart-Homes verwirklicht und weitergeführt, wie z. B. im Living-Place-Projekt ([von Luck u. a., 2010](#)) der HAW Hamburg.

## 1.1. Motivation

Digital Natives verweben ganz selbstverständlich ihr Privatleben mit der Onlinewelt. Sie pflegen online Freundschaften, schließen Verabredungen, kaufen online ein, suchen online nach Informationen aller Art, speichern ihre Adressbücher, Termine und andere persönliche Daten in der Cloud.

Die Technik im Alltag, vor allem die Technik in unseren Wohnräumen, soll den Menschen unterstützen. Idealerweise weiß die Wohnung schon, was als nächstes benötigt wird. Kündigen sich Freunde zu Besuch an, wäre es wünschenswert, wenn die Wohnung den Bewohner im

Vorfeld fragen würde, ob er den Gästen irgendetwas anbieten möchte und wenn ja, dass die Wohnung den Bewohner darauf hinweist, dass dafür noch Lebensmittel fehlten.

Immer leistungsfähigere und günstiger werdende Computer und Kommunikationstechnologie bietet die Möglichkeit, die Technik weniger als Werkzeug zu gestalten, sondern viel mehr als Unterstützer zu designen, der als bessere Hälfte oder „Freund“ dem Besitzer Hinweise gibt, sollte er was vergessen, oder Dinge für den Besitzer erledigt und vorbereitet.

In intelligenten Wohnungen sind andere Interaktionsmöglichkeiten zwischen Mensch und Maschine jenseits von Maus und Tastatur notwendig, wie in Selker (2007) einem Seminar über Context-Aware-Computing gezeigt wird.

### 1.2. Zielsetzung

Das Ziel des Smart-Home Forschungsbereichs ist ein System, das sozusagen wie ein guter Freund agiert und den Menschen bei der Alltagsbewältigung unterstützt, an Termine, Einkaufen und andere Dinge erinnert oder Lösungen anbietet.

Der Sonderforschungsbereich Transregio 62 (sfbtransregio:62), an dem die Universität Ulm, die Otto-von-Guericke Universität Magdeburg und das Leibniz-Institut für Neurobiologie beteiligt sind, formuliert seine Vision für die Erforschung neuartiger technischer Systeme im Hinblick auf die Mensch-Maschine-Interaktion folgendermaßen:

„Das Forschungsvorhaben folgt der Vision, dass technische Systeme der Zukunft Companion-Systeme sind – kognitive technische Systeme, die ihre Funktionalität vollkommen individuell auf den jeweiligen Nutzer abstimmen: Sie orientieren sich an seinen Fähigkeiten, Vorlieben, Anforderungen und aktuellen Bedürfnissen und stellen sich auf seine Situation und emotionale Befindlichkeit ein. Dabei sind sie stets verfügbar, kooperativ und vertrauenswürdig und treten ihrem Nutzer als kompetente und partnerschaftliche Dienstleister gegenüber.“

Diese Beschreibung stellt den visionären Überbau dar, unter dem sich die Ziele, die die vorliegende Arbeit betreffen, versammeln.

Konkret auf die vorliegende Arbeit bezogen, ist ein System das Ziel, das in einem Smart-Home Lebenssituationen automatisch erkennt und adäquat auf diese reagiert. Es soll die Lebenssituationen nicht nur erkennen, sondern darüber hinaus vorhersehen, was als nächstes von Bedeutung sein könnte, um den Menschen bestmögliche Unterstützung anbieten zu können.

Die Architektur soll dem Multiagentenansatz folgen um ein flexibles und modulares System zu ermöglichen, so dass neue Funktionalitäten und Eigenschaften durch neue Agenten dynamisch

hinzu gefügt werden können. Ziel dieser Arbeit ist es ein Modell eines Systems zu entwickeln, das in einer bekannten Wohnsituation Handlungen und Ereignisse voraussieht, um den (intelligenten) Geräten in der Wohnung erwartete Zielzustände mitteilen zu können. Am Ende der Arbeit soll auf Grundlage dieses Modells ein erster Prototyp entwickelt werden, anhand dessen Teile der theoretischen Überlegungen überprüft werden können. Durch zwei Alltagsszenarien werden die Überlegungen gestützt: Frühstückssituation und Besuch von Freunden.

Die Interpretation dieser Szenarien wird für die vorliegende Masterarbeit als gegeben vorausgesetzt und in der parallel entstehenden Arbeit von [Ellenberg \(2011b\)](#) behandelt.

### 1.3. Aufbau der Arbeit

Für den Einstieg in die Thematik des Smart-Home und Context-Aware-Computing werden in den **Grundlagen** (Kapitel 2) zunächst damit eng verbundene Begriffe erläutert (Abschnitt 2.1). Anschließend wird ein Überblick über den Stand der Forschung im Bereich des Smart-Homes und des Context-Aware-Computing gegeben (Abschnitt 2.2). Abschließend werden die beiden technologischen Themen „Automatisches Planen“ und „Zeit als Ressource“ umrissen (Abschnitt 2.4 und Abschnitt 2.5), die eine zentrale Rolle bei Szenarien basierten Interpretationen in einem Smart-Home spielen.

In der **Analyse** (Kapitel 3, Abschnitt 3) wird anhand von zwei Userstories (Abschnitt 3.1) die Definition und Darstellung von Zielen (Abschnitt 3.2), sowie deren Zusammenschluss zu Szenarien erarbeitet (Abschnitt 3.3).

Im Kapitel **Design und Realisierung** (Kapitel 4) wird in der ersten Hälfte ein Modell eines GoalManagers entwickelt, der mittels Graphensuche und Zeitintervallalgebra zu erreichende Ziele auswählt und an GoalAgents weiter gibt. In der zweiten Hälfte wird aus diesem Modell ein Prototyp entwickelt (ab Abschnitt 4.7), mit dessen Hilfe einige der erarbeiteten Bedingungen überprüft werden (ab Abschnitt 4.10).

Den Schluss bildet das Kapitel **Zusammenfassung und Ausblick** (Kapitel 5), in dem die Ergebnisse dieser Arbeit zusammengefasst und bewertet werden (Abschnitt 5.1), sowie ein Ausblick (Abschnitt 5.2) darauf gegeben wird, wie weiter an der Thematik der Szenarien basierten Interpretation eines Smart-Homes geforscht werden kann.

## 2. Grundlagen

In diesem Kapitel werden zum einen bestimmte Aspekte des Themenkomplexes Smart-Home erläutert, zum anderen in spezielle technische Themen eingeführt, die später in dieser Arbeit Verwendung finden.

Im Abschnitt 2.1 werden Grundbegriffe zum Thema Intelligente Wohnungen erläutert. Anschließend werden im Abschnitt 2.2 Projekte aus diesem Bereich vorgestellt. Es wird explizit darauf eingegangen, in wie weit die Zielstellungen der Projekte das Thema dieser Arbeit betreffen und welche Ergebnisse diesbezüglich erzielt wurden.

Zudem werden Grundlagen im Automatischen Planen (Abschnitt 2.4) und zum Aspekt Zeit als Ressource (Abschnitt 2.5) vermittelt.

### 2.1. Intelligente Wohnungen

Aus dem Bedürfnis heraus die den Menschen umgebene Technik in der Bedienung mehr homogen und intuitiv zu gestalten, und der Notwendigkeit die Technik energieeffizient zu betreiben, sind rund um die Welt verschiedene Projekte entstanden, die im Themenkontext der intelligenten Wohnungen forschen. Eine Spezialisierung der intelligenten Wohnungen ist das Ambient Assisted Living, in dem alte Menschen in ihrem Alltag durch die Wohnung unterstützt werden sollen. Eine andere Spezialisierung ist das Home Office, das eine Erweiterung der Smart-Home Umgebung darstellt. Dort wird der reine Wohnraum zu einem kombinierten Wohn- und Heimarbeitsplatz ausgeweitet. Das Smart-Home soll dabei flexibel zwischen Arbeits- und Wohnsituation wechseln können, indem es Lichtstimmungen, dynamische Schaltflächen, Anruffilter usw. nach Bedarf steuert und verändert.

#### 2.1.1. Ubiquitous Computing

Ubiquitous-, Pervasive- oder auch Disappearing computing bezeichnet den Trend, Computer nach und nach in den Alltag zu integrieren, so dass sie nicht mehr als einzelne technische Einheiten wahrgenommen werden. Beispiele sind die computergesteuerte Waschmaschine, Fernsehgeräte

und Festplatten-Videorekorder, moderne Autos mit ihren Boardsystemen, wie ABS, ASR und Motorregelung, Smartphones etc.

Dabei verändert sich die Bedienung der Geräte nur unwesentlich:

Die Waschmaschine besitzt immer noch einen Drehschalter für die Programmwahl. Dahinter arbeitet inzwischen allerdings ein computergestützter, mit elektronischer Sensorik ausgestatteter Steuerautomat. Die Motoren von modernen Autos werden von einem Boardcomputer überprüft, der Wartungszyklen kontrolliert, Ölstand und Verbrauch misst sowie die Zündung regelt.

In modernen Fernsehgeräten wird in Echtzeit das aktuelle Fernsehprogramm, das digital übertragen wird, dekodiert. Möchte man den Fernsehsender wechseln, so muss man nicht mehr den Tuner direkt umschalten, sondern weist den Computer im Fernseher an, einen anderen Kanal zu dekodieren. Zudem arbeiten die meisten Videorekorder nur noch per Festplatte und bieten Timeshift<sup>1</sup> an. Außerdem trägt man für eine Aufnahme keine Zeiten mehr in den Videorekorder ein, sondern sucht sich die gewünschte Sendung aus der elektronischen Programmübersicht und markiert diese zur Aufnahme, welche selbstständig startet und zur rechten Zeit endet.

Der Mensch nimmt die ihn umgebenden Computer nicht mehr als solche wahr, obwohl sie maßgeblich zu seinem Alltag gehören. Eine Einführung in den Themenkomplex des Ubiquitous und Persavice Computing findet sich in [Moran und Dourish \(2001\)](#).

### 2.1.2. Kontext und Context-Awareness

Interaktionen zwischen Menschen oder Mensch und Technik sind oft abhängig vom Kontext. Nach dem morgendlichen Aufstehen ins Badezimmer zu gehen hat einen anderen Hintergrund als der gleiche Gang abends während des Fernsehguckens oder kurz vor dem Schlafengehen. Menschen bringen von sich aus Handlungen in den richtigen Kontext, wenn sie mit der Situation aus ihrem Alltag vertraut sind. Heutigen Computersystemen ist dieser Kontext in der Regel noch nicht bekannt.

Ein Handy klingelt immer, sobald ein Anruf eintrifft, unabhängig davon, ob sich der/die Besitzer/Besitzerin in einem Gespräch oder Meeting befindet, an der Kasse im Supermarkt steht oder sich mit dem Auto fahrend auf den Straßenverkehr konzentrieren muss.

Um eine Vorstellung davon zu bekommen, was aus technischer Sicht Kontext bedeutet wird in [Abowd u. a. \(1999\)](#) der Kontext wie folgt definiert:

„We define context as any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or physical or computational

---

<sup>1</sup>zeitversetztes Fernsehen

object. We define context-awareness or context-aware computing as the use of context to provide task-relevant information and/or services to a user.“

Der Kontext beinhaltet also alle Informationen, die verwendet werden können, um die aktuelle Situation zu beschreiben. Context-Awareness beschreibt Computersysteme, die diese Kontextinformationen verwenden, um den Benutzer kontextbezogene Informationen und Unterstützung anzubieten.

Aus technischer Sicht gibt es verschiedene Ebenen des Kontextes. Vereinfacht können diese Ebenen in primitiver Kontext und höherer Kontext eingeteilt werden. Der primitive Kontext ergibt sich aus direkter Transformation von Sensorwerten in Aussagen, z. B. die Uhrzeit und Datum, ist es Tag oder Nacht, die Information ob jemand im Raum ist oder nicht. Höherer Kontext entsteht durch Interpretationen des primitiven Kontextes und führt zu Aussagen wie z. B. ob jemand wach ist oder nicht, jemand sich ein Mahlzeit zubereitet, liest oder fernsieht.

In [Greenberg \(2001\)](#) wird zudem betont, dass Kontext ein dynamisches Konstrukt ist:

„Context is a dynamic construct. Although some contextual situations are fairly stable, discernable, and predictable, there are many others that are not. Similar looking contextual situations may actually differ dramatically, due perhaps to people’s previous episodes of use, the state of their social interactions, their changing internal goals, and the nuances of local influences.“

Da viele Situationen ähnlich erscheinen, aber völlig unterschiedliche Bedeutung haben können und es daher ein komplexes Vorhaben ist auf diskrete Zustände zu schließen und daraus bestimmte Aktionen abzuleiten, ist es schwierig ein System zu designen, das context-aware ist.

### 2.2. Stand der Forschung

Im Folgenden werden Projekte und Arbeiten zum Thema Smart-Home und dem Bereich Kontexterkenkung vorgestellt und ein grober Überblick über die technischen Lösungsstrategien gegeben.

Als erstes werden die Projekte *ubiHome* (Abschnitt 2.2.1), *Personal Home Server* (Abschnitt 2.2.2) und *HomeLab* (Abschnitt 2.2.3) aus dem Smart Home Bereich umrissen. Anschließend wird das Projekt *CoFriend* (Abschnitt 2.2.4) vorgestellt, das als Ziel die Erkennung einer Flugzeugabfertigung hat. Erkenntnisse dieses Projekts beeinflussten die vorliegende Arbeit maßgeblich. Als letztes wird kurz auf die Diplomarbeit von [Gottmann und Nachtigall \(2011\)](#) eingegangen, in welcher das Living Place als höheres Petrinetz modelliert wird, um dessen Funktionalität formal nachzuweisen.

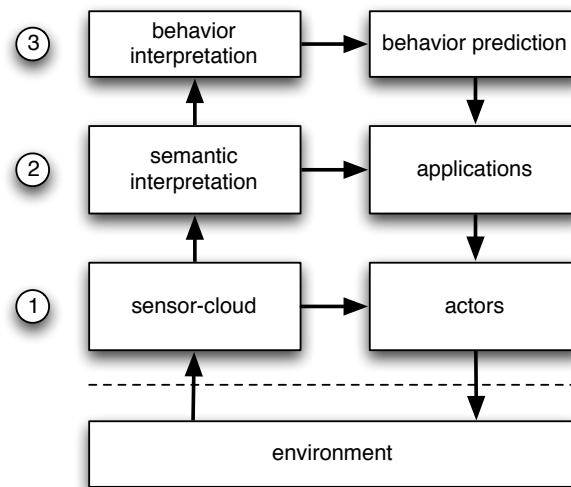


Abbildung 2.1.: Abstraktionsebenen der Kontextverarbeitung zur Kategorisierung der Smart-Home Projekte

Abbildung 2.1 zeigt ein Schaubild, in dem die Struktur eines kontextsensitiven Systems in Abstraktionsebenen und Funktionsblöcke unterteilt wird. Ebene eins befasst sich mit der technischen Umsetzung, die Umwelt mittels Sensoren wahrzunehmen, sie über Aktoren zu beeinflussen und diese Funktionalität als Dienste im Smart-Home zur Verfügung zu stellen. Auf Ebene zwei werden Dienste integriert, die über die rohen Sensorwerte semantische Aussagen bilden und Anwendungen bereitstellen, die diese Aussagen verwenden und anschließend Aktoren steuern. Die semantischen Aussagen auf dieser Ebene sind isoliert voneinander. Die Aussage „der Bewohner befindet sich seit 15 Minuten in der Küche“ wird z.B. nicht damit in Verbindung gebracht, dass der Bewohner sich auf einen Besuch vorbereitet und deshalb gleich ein Essen zubereiten möchte. Ebene drei schließt den Kreis und verknüpft einzelne semantische Aussagen zu Aktivitäten und ganzen Szenarien.

Die Projekte *ubiHome* (Abschnitt 2.2.1), *Personal Home Server* (Abschnitt 2.2.2) und *Home-Lab* (Abschnitt 2.2.3) sind in den Ebenen eins bis zwei angesiedelt, während sich das *CoFriend* Projekt (Abschnitt 2.2.4) auf Ebene drei bewegt.

### 2.2.1. ubiHome

Das *ubiHome* Projekt am Electronics and Telecommunications Research Institute in Korea hat sich mit der Entwicklung einer Smart-Home Infrastruktur beschäftigt. In Ha u. a. (2007) werden die Ergebnisse hierzu vorgestellt. Der Fokus dieser Infrastruktur liegt einerseits auf der technischen



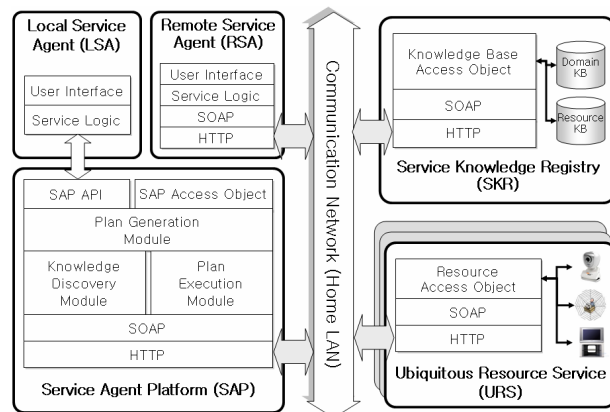


Abbildung 2.2.: Architektur-Übersicht des ubiHome Projekts (Ha u. a., 2007, Fig. 2)

Umsetzung der Kommunikation zwischen den verschiedenen Geräten und andererseits auf dem Auffinden der passenden Services zu Befehlen wie „Become Brighter“.

Die Architektur (Abbildung 2.2) von ubiHome ist agentenorientiert. Im Zentrum steht die Service Agent Plattform (SAP), welche den Agenten als Middleware dient und diese koordiniert. Unterschieden werden lokale Serviceagenten (LSA) und entfernte Serviceagenten (RSA). Erstere befinden sich auf der gleichen Maschine wie die SAP und kommunizieren direkt über die API mit der SAP. Die entfernten Agenten befinden sich auf anderen Maschinen als die SAP und kommunizieren mit dieser über das LAN der Wohnung per Web Services. Beide Arten von Agenten besitzen ein Userinterface und eine Servicelogik, mit der sie Anwendungsfälle bereitstellen.

Alle Services werden mit OWL-S<sup>2</sup> als Ontologie beschrieben. Der Ubiquitous Resource Service (URS) bietet Web-Services für jede Resource der Wohnung an. In der Service Knowledge Registry (SKR) ist das Wissen hinterlegt, welche Ressource zur Erfüllung von Befehlen bzw. Aufgaben verwendet werden kann.

Wird nun der Befehl „Become Brighter“ an das System gestellt, d. h. das Licht in dem Zimmer, in dem sich der Bewohner befindet, soll heller werden, dann wird nach den passenden Services gesucht (z. B. „getUserLocation“) und ein Plan erstellt, was zu machen ist, um das Licht im richtigen Raum heller werden zu lassen. Als Folge des Befehls können z. B. entweder die Deckenlampen heller gestellt werden, die Jalousien aufklappen oder ein Roboter die Stehlampen einschalten.

<sup>2</sup>Web Ontology Language for Webservices, siehe dazu Martin u. a. (2006)

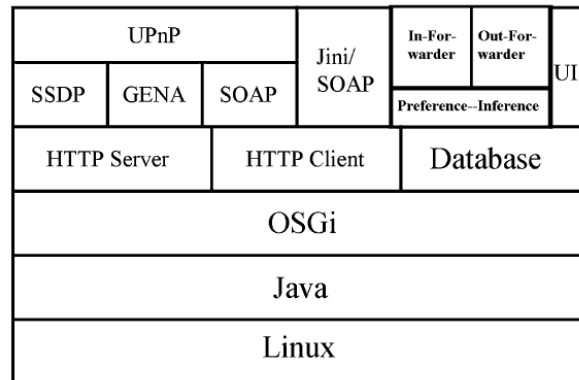


Abbildung 2.3.: Architektur-Übersicht des Personal Home Servers (Nakajima und Satoh, 2006)

Dieser Plan wird mit einem HTN<sup>3</sup> Planer erstellt und anschließend in BPEL4WS<sup>4</sup> transformiert. Der in BPEL4WS vorliegende Plan kann dann vom System ausgeführt werden.

Entscheidend bei dieser klassischen Planung ist, dass der Planungsprozess erst beginnen kann, wenn alle Services aufgefunden wurden und bekannt sind. Außerdem wird immer ein vollständiger Ablaufplan erstellt, ändert sich der Kontext während des Planungsprozesses, muss der ganze Prozess erneut gestartet werden. Das hat zur Folge, dass der ganze bisher erstellte Plan verworfen werden muss, selbst wenn Teile des alten Plans theoretisch auch im neuen Plan verwendet werden könnten.

### 2.2.2. Personal Home Server

An der Waseda Universität in Japan haben Nakajima und Satoh (2006) einen Personal Home Server entwickelt, der auf jedem Gerät einer intelligenten Umgebung laufen soll. Dieser Server soll für den Einsatzzweck auf dem jeweiligen Gerät und die Person personalisiert werden, zusätzlich stellt er die Services dieses Geräts, den Geräten in der Umgebung zur Verfügung.

Abbildung 2.3 zeigt die Architektur des Systems. Augenmerk ist bei dieser Lösung die technische Umsetzung Services dynamisch zu finden und anzubieten. Dazu kommen in diesem Projekt Techniken und Protokolle wie u. a. UPnP (Universal Plug and Play), Jini<sup>5</sup> und SOAP<sup>6</sup> zum Einsatz. Die Services werden mit der weithin gebräuchlichen RDF (Resource Description

<sup>3</sup>Prinzip eines „Hierarchical Task Network“-Planers in Nau u. a. (2004) S. 229ff

<sup>4</sup>Business Process Execution Language für WebServices

<sup>5</sup>Framework zum Programmieren von verteilten Anwendungen

<sup>6</sup>Ursprünglich für Simple Object Access Protocol, ist ein XML basiertes Netzwerkprotokoll für Remote Procedure Calls

Framework) beschrieben. Die einzelnen Services können über eine URL (Uniform Resource Locator) angesprochen werden. In SWI-Prolog wurde eine Inerence Engine implementiert, die zu einer angeforderten Ressource, abhängig von Rahmenparametern, die passenden Services findet. Ein Prolog Term um einen Service aufzufinden kann folgendermaßen aussehen (Nakajima und Satoh, 2006, Seite 384):

```
1 Function(x, TV) := Location(x, living-room), Support(x, BS).
2 Function(x, Light) := Type(x, ceiling), Location(x, y),
3                     Location(me, y), Room(y).
```

Nach der in Abbildung 2.1 aufgestellten Aufteilung in die drei Abstraktionsebenen bewegt sich dieses Projekt prinzipiell noch unterhalb der ersten Ebene. Der Fokus liegt hier auf der technischen Lösung der grundlegenden Probleme für das dynamische Auffinden und dynamische Anbieten von Services in einer heterogenen Umgebung.

### 2.2.3. HomeLab

An dem unter der Leitung von Philips Research entstandenem Amigo Projekt (Janse, 2008) wirkten verschiedene europäische Firmen und Universitäten mit. Das Ziel des Projekts bestand darin eine Plattform für das intelligente Haus zu schaffen. Im Rahmen dieses Projektes und einer Doktorarbeit (Vallée, 2009)<sup>7</sup> plante Vallée ein Multi-Agenten-System (Vallée u. a., 2005a) einzusetzen, das eine dynamische Service-Komposition ermöglicht.

Vallées System arbeitet mit Jade-Agenten, die jeweils eine Komponente des Gesamtsystems darstellen. Diese Agenten werden zu Teams zusammengestellt, die jeweils eine Anwendung bzw. einen Systemaspekt abbilden. Spezielle Vermittleragenten übernehmen die teamübergreifende Koordinierung (siehe Anhang F.1, S. 106). Vallée hat sich gegen einen Workflowansatz mit Planning entschieden, da seine Versuche ergeben haben, dass dieses Verfahren bei kontinuierlichen Kontextwechseln zu schwer zu handhaben wäre und zu langsam arbeiten würde.

Diese Erfahrungen geben einen Hinweis darauf, dass Workflows, die auch als ein Abbildung von Szenarien angesehen werden können, zu unflexibel in dynamischen Umgebungen sind. Für die Grundfunktionalität in einem Smart-Home ist ein adaptives System, in dem jede Komponente ihre Aufgabe besitzt, praktikabler.

### 2.2.4. CoFriend - Behaviour-Interpretation

Eine Spezialform der Context-Awareness ist die Behaviour-Interpretation. Als Beispiel dient das CoFriend-Projekt aus Bohlken u. a. (2011).

---

<sup>7</sup>da die Arbeit auf Französisch ist, siehe Anhang F.1 S. 106

CoFriend war ein europäisch gefördertes Forschungsprojekt an dem unter anderen die Universität Hamburg beteiligt war. Es beschäftigte sich mit der Erkennung einer kompletten Abfertigung eines Passagierflugzeugs auf dem Rollfeld vor dem Terminal.

Mit Hilfe von sechs Kameras wurde der Abfertigungsplatz beobachtet und das Geschehen in Echtzeit von einem System ausgewertet. Das komplette System arbeitet in drei Ebenen. Anhand des Bildmaterials werden auf der untersten dieser drei Ebenen Objekte und deren Aufenthaltsorte erkannt. Diese Informationen dienen der nächst höheren Ebene dazu Ereignisse zu erkennen, wie z.B. *Airplane enters zone*. Diese Ereignisse dienen der obersten Ebene als Evidenz zur Erkennung bestimmter Abläufe der Abfertigung. Die Universität Hamburg war für die Entwicklung der obersten Ebene verantwortlich und beschreibt ihre Ergebnisse in [Bohlken u. a. \(2011\)](#).

Auf dieser obersten Ebene wird das Abfertigungsszenario als Ontologie beschrieben. Diese ist um Zeitregeln auf Basis von Allens Intervallalgebra ergänzt worden. Dazu diente die Ontologiebeschreibungssprache OWL und die Semantic WebRule Language SWRL. Diese Art der Beschreibung ermöglicht eine abstrakte implementationsunabhängige Beschreibung des Gesamtszenarios.

Das Upper-Model der Ontologie, das heißt das domänenunabhängige Modell, zeigt die Abbildung [2.4](#). Sie besteht aus zwei Hauptgruppen von Objekten, zum einen die konzeptionellen Objekte (Conceptual Object) und zum anderen die physikalischen Objekte (Physical Object). Letztere unterteilen sich in mobile Objekte (Fahrzeuge und Menschen) und unbewegliche Objekte (Bereiche der Abfertigungszone).

Zu den konzeptionellen Objekten gehören Zustände (State) und Ereignisse (Event). Wie der Begriff Zustand nahelegt, handelt es sich dabei um Aussagen wie z.B. das „Flugzeug wird betankt“, das heißt, es handelt sich um Konzepte, die über eine gewisse Zeit andauern. Im Gegensatz dazu die Ereignisse, die zu einem Zeitpunkt auftreten, beispielsweise „Tanker enters zone“.

Unterschieden wird in primitive Ereignisse und Zustände und in zusammengesetzte Ereignissen und Zustände. Primitive Ereignisse und Zustände sind die Blätter in der Ontologie, die später im Interpretationsprozess mit der Auswertung der Bilderkennung in Verbindung gebracht werden. Sie stellen die so genannte Evidenz dar. Während der Laufzeit wird also das wirklich erkannte Event „Tanker stopped inside zone“ aus der Bilderkennung mit dem Konzept „Tanker stopped inside zone“ aus der Ontologie verbunden und somit eine Evidenz hergestellt. Zusammengesetzte Objekte, die auf primitiven Objekten aufbauen, können als wahr angesehen werden, wenn alle ihre primitiven Kindobjekte mit Evidenz belegt werden. Diese Evidenz wird in der Hierarchie nach oben gereicht, sofern die Kinder höherer Objekte als wahr gelten. Ereignisse und Zustände werden zu Aktivitäten zusammen gesetzt, in diesem Beispiel gehören die Events zur „Flugzeug wird betankt“ Aktivität. Alle Aktivitäten zusammen genommen, bilden die Flugzeugabfertigung

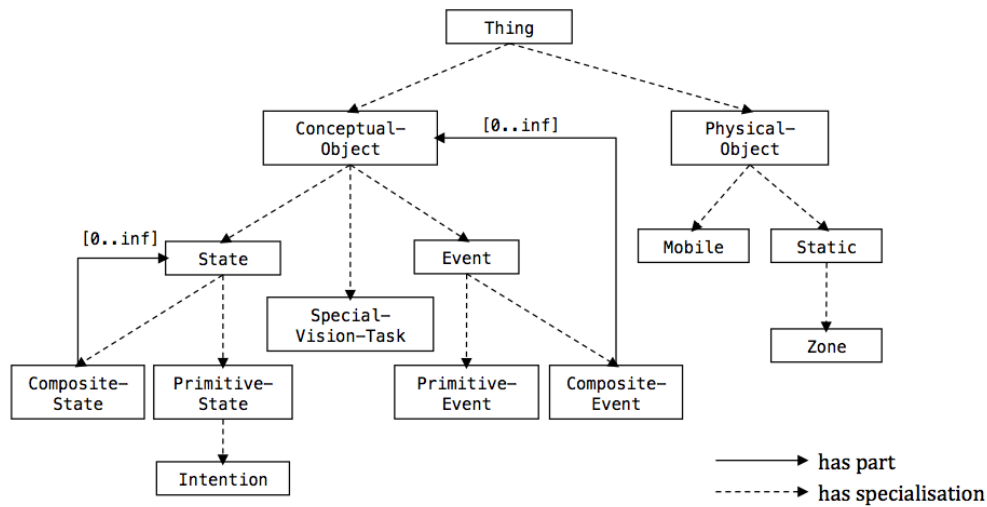


Abbildung 2.4.: Upper model aus (Bohlken u. a., 2011)

ab. Für eine korrekte Flugzeugabfertigung müssen die Aktivitäten einer gewissen zeitlichen Ordnung auftreten, diese wird über die Zeitalgebra von Allen<sup>8</sup> definiert.

Im SCENIOR-System der Universität Hamburg findet kein direktes Reasoning über der Ontologie statt. Die Ontologie dient ausschließlich zur Beschreibung der Domäne. Damit SCENIOR arbeiten kann, wird die Ontologie mittels eines Konverters in verschiedene Bestandteile überführt. Unter anderem in Regeln für eine Regelmaschine<sup>9</sup>, in einen Hypothesengraphen und in ein Temporal Constraint Net, das die Informationen über die zeitlichen Beziehungen zwischen den Aktivitäten enthält.

Auch wenn das CoFriend-Projekt eine Flugzeugabfertigung behandelt, sind die Prinzipien der Behaviour-Interpretation ähnlich denen in einem Smart-Home. Das Projekt zeigt einen Weg, wie Szenarien in einem Smart-Home abgebildet werden könnten. Besonders die Aufteilung in Aktivitäten und deren relative Ordnung über die Intervallalgebra von Allen sind hier hervorzuheben.

### 2.2.5. Living Place als Petrinetz

Gottmann und Nachtigall (2011) bilden in ihrer Diplomarbeit mit Hilfe von höheren Petrinetzen die Funktionalität des Living Places der HAW nach. Sie modellieren, wie Geräte sich und ihre Services im Living Place bekannt machen können, und wie die Services von anderen Geräten und

<sup>8</sup>siehe Abschnitt 2.5

<sup>9</sup>JESS: Java Expert System Shell

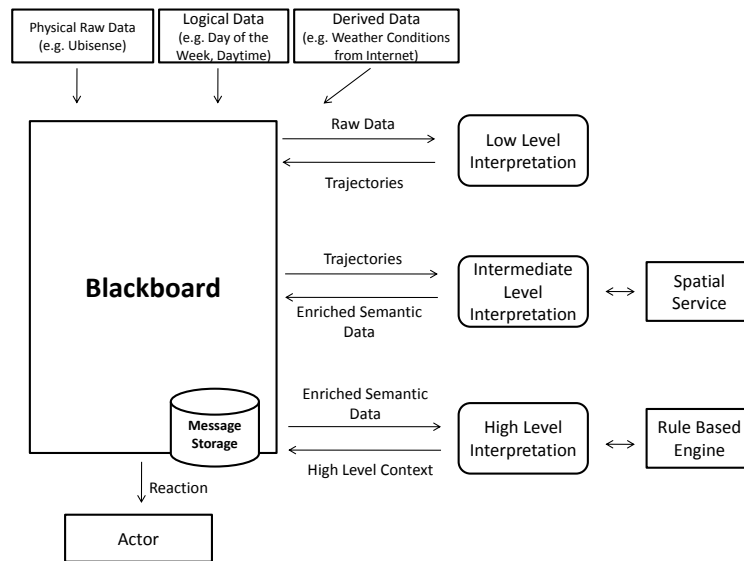


Abbildung 2.5.: Architekturskizze des Living Places aus [Ellenberg u. a. \(2011\)](#)

Services aufgefunden werden. Ihnen ist es gelungen ein vollständiges Modell zu entwickeln und zu zeigen, dass dieses Netz schalten kann. Da es noch nicht automatisiert überprüft werden kann, stellt dies erst einen Anfang dar, die Funktionalität eines Smart-Homes oder die Funktionalität von Teilsystemen eines Smart-Homes formal nachzuweisen.

### 2.3. Architektur des Living Places

In [Ellenberg u. a. \(2011\)](#) wird die grundlegende Architektur des Living Places beschrieben, diese wird in [Abbildung 2.5](#) skizziert. Zentraler Teil der Architektur ist ein Blackboard, welches die Koordination der einzelnen Komponenten gewährleistet. Der Blackboard-Begriff wurde von [Erman u. a. \(1980\)](#) eingeführt. Das Blackboard stellt eine zentrale Systemkomponente dar, auf der z. B. Aufgaben hinterlegt und gemeinsam von verschiedenen Agenten bearbeitet werden können. So wird es auch im Living Place gehandhabt. Auf dem Blackboard laufen alle Informationen zusammen und jede Komponenten, die diese Informationen benötigt, kann sie sich vom Blackboard holen.

Die zur Zeit vorhandenen und in Bearbeitung befindlichen Komponenten, wie sie für die vorliegende Arbeit vorausgesetzt werden sind in [Abbildung 2.6](#) dargestellt.

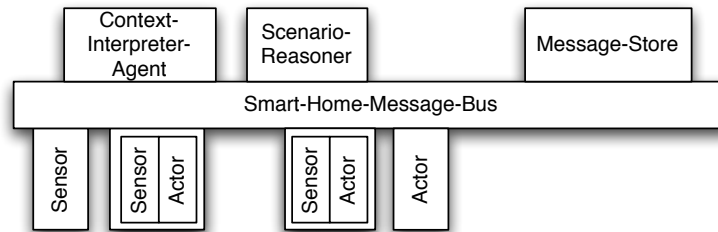


Abbildung 2.6.: Architektur des Living Places

Sensoren und Aktoren bilden die allgemeine Infrastruktur des Living Places. Sensoren gehören zu bestimmten logischen Einheiten, die Werte aus ihren Wertedomänen zur Verfügung stellen. Aktoren können Aktionen ausführen und den Zustand des Systems verändern. Sie sind die Geräte im Living Place oder andere Softwarekomponenten (z. B. ein Wetterinformationsagent) und liefern aus Sicht der vorliegenden Arbeit low level Kontextinformationen oder können Steuerbefehle empfangen und diese in Aktionen umsetzen.

Verbunden werden alle Komponenten über einen System Bus, ähnlich einer SOA<sup>10</sup> werden sie auf diese Weise lose miteinander gekoppelt. Anders als in der SOA, bieten die Komponenten keine Services über Schnittstellen an, sondern tauschen Nachrichten miteinander aus, auf die sie jeweils reagieren können. Genauer ist [Ellenberg u. a. \(2011\)](#) zu entnehmen.

### 2.4. Automatisches Planen

Bei Computersystemen die selbstständig Probleme lösen sollen und/oder dynamisch auf ihre Umgebung reagieren müssen, werden u. a. die Methodiken des automatischen Planens angewendet. Automatisches Planen arbeitet mit Ressourcen und Aktionen, die die Ressourcen verändern können. Betrachtet man das automatische Planen als Zustandstransitionssystem (Abbildung 2.7), besteht es aus dem Quadrupel ([Nau u. a., 2004](#), Seite 5/6):

$$(S, A, E, \gamma)$$

- $S = s_1, s_2, \dots$  ist eine Menge an Welt-/Systemzuständen,
- $A = a_1, a_2, \dots$  ist eine Menge an Aktionen,
- $E = e_1, e_2, \dots$  ist eine Menge an Ereignissen und

---

<sup>10</sup>Service Orientated Architecture

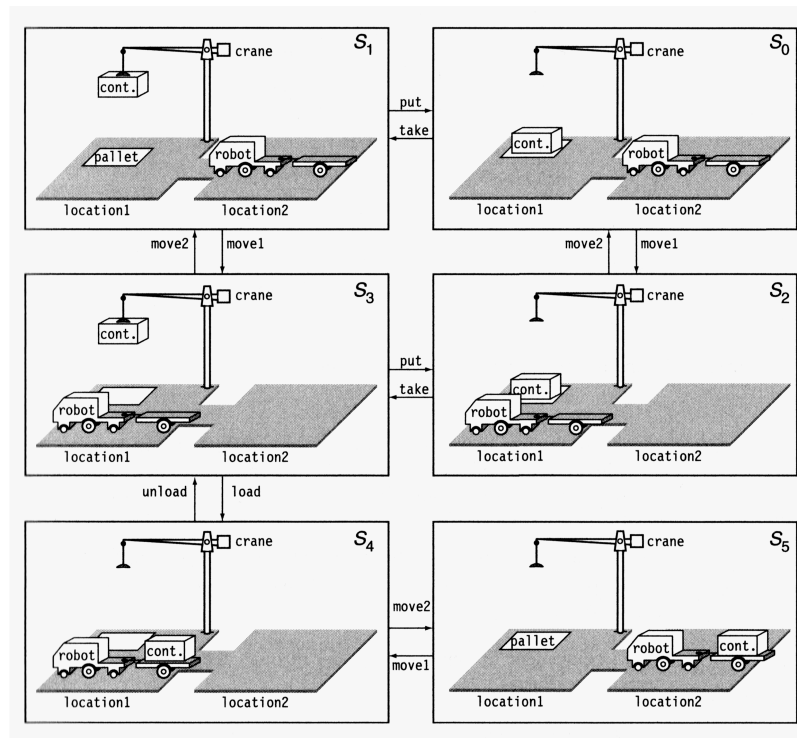


Abbildung 2.7.: Planning als State-Transition-System betrachtet (Nau u. a., 2004, S. 7f). Von den einzelnen Zuständen aus gibt es Transitionen, die in einem anderen Zustand führen. Nicht jeder Zustand ist gleichermaßen zu erreichen.

- $\gamma : S \times A \times E \rightarrow 2^S$  ist eine Zustandstransitionsfunktion.

Diese Sicht auf das automatische Planen definiert für das zu beschreibende System diskrete Zustände ( $S$ ), von denen über eine Zustandstransitionsfunktion ( $\gamma$ ) mittels Ereignissen ( $E$ ) und Aktionen ( $A$ ) in einen nächsten Zustand gewechselt werden kann. Prinzipiell sind die Ereignisse und Aktionen dimensionslos, d. h. deren Wirkung ist unmittelbar. Dieses Konzept wird in Abbildung 2.8a dargestellt.

Der Planer besitzt ein Welt-/Systemmodell und erstellt auf Grundlage eines Ausgangszustandes des Systems und den zu erreichenden Zielen einen Plan von Aktionen. Misslingt die Ausführung des Planes aus irgendwelchen Gründen, erhält der Planer darüber keine Informationen. Erst im nächsten Planungszyklus stellt der Planer fest, dass nicht alle Ziele erreicht wurden. Auf Basis des neuen Systemzustandes und möglicherweise veränderter Ziele muss er einen neuen Plan erstellen.



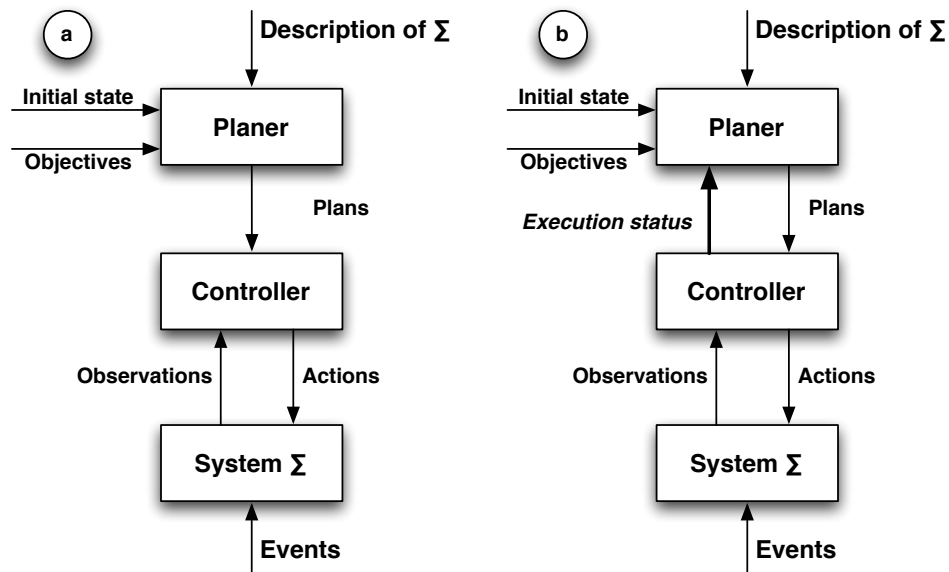


Abbildung 2.8.: a) einfacher Planer, b) dynamischer Planer (Nau u. a., 2004, S. 8f)

Das Konzept des dynamischen Planers soll diesen Nachteil ausgleichen (Abbildung 2.8b). Der Controller kann beim dynamischen Planen dem Planer bei Abweichungen Rückmeldung geben. Anschließend wendet der Planer Hilfs- und Repairpläne an, um das System wieder in einen Zustand zu bringen, von dem aus er die noch offenen Ziele mit einem neuen partiellen Plan erreichen kann.

Problematisch sind allerdings Ereignisse, die nicht im Modell des Systems definiert sind. Diese können durch den Planer nicht berücksichtigt werden. Pläne die aufgrund dieser Ereignisse scheitern, wird der Planer nicht korrigieren können. Das Gleiche gilt für Seiteneffekte von Aktionen, die das System auf eine Art verändern, die im Modell nicht berücksichtigt wird (Ramification-Problem). Das heißt, nach dem Ausführen eines Plans, befindet sich das System in einem anderen Zustand, als von dem Planer angenommen.

Ein anderes Problem ist das Frame-Problem. Als Beispiel: Es ist nicht automatisch klar, ob eine Aktion, die bewirkt, dass sich ein Fahrzeug  $F$  von Ort  $O_1$  nach Ort  $O_2$  bewegt, auch beinhaltet, dass sich die Ladung auf dem Fahrzeug von Ort  $O_1$  nach Ort  $O_2$  bewegt. In Abbildung 2.7 ist dies in Zustand  $S_4$  und  $S_5$  der Fall. In geschlossenen und vollständig bekannten Systemen können solche Probleme ausgeschlossen werden, zum Beispiel gilt das für einen Roboter, der sich in der normalen Umwelt bewegt, nicht. Es könnte auch als das Problem benannt werden, den gesunden Menschenverstand oder das Verständnis für naive Physik in der Maschine zu implementieren.

Klassische Planer sind in der Regel zentralisiert, es gibt einen zentralen Planer, der einen Plan erstellt, der von einem Controller ausgeführt wird. Viele Aufgaben ermöglichen es jedoch das Planen aufzuteilen oder die Ausführung des Plans aufzuteilen. Neben Ansätzen wie DCSP<sup>11</sup> (Nau u. a., 2004, CSP: Seite 19ff, 169ff) und HTN<sup>12</sup> (Nau u. a., 2004, Seite 229ff), gibt es auch Ansätze mit Multiagenten-Systemen (Nau u. a., 2004, Seite 530ff). Beim Multiagent-Planning gibt es verschiedene Varianten, entweder zentrale Planung und verteilte Ausführung oder verteilte Planung und verteilte Ausführung (de Weerd u. a., 2005).

In der verteilten Planung haben die einzelnen Agenten neben den übergeordneten Zielen auch eigene Ziele, die sie verfolgen. Die Schwierigkeit beim verteilten Multiagenten-Planen ist die Koordination zwischen den Agenten, so dass das Gesamtsystem zu einem befriedigenden Ergebnis gelangt.

Zusammengefasst geht es bei dem automatischen Planen darum mit gegebenen Ressourcen und Aktionen von einem Ausgangszustand zu einem Zielzustand zu gelangen. Dazu werden Aktionspläne erstellt, die jeden Schritt von einem Ausgangszustand zum Zielzustand beinhalten. Die Schwierigkeit in dieser Form des Planens ist die Fehlerbehandlung und generell die Komplexität des Planproblems.

### 2.5. Zeit als Ressource

Im letzten Abschnitt wurde das automatische Planen erläutert, bei dem Aktionen und Ressourcen geplant werden. Wie in dem Beispiel in Abbildung 2.7, in dem Krahn, LKW und Ladung als Ressourcen betrachtet werden, kann auch die Zeit als Ressource betrachtet werden.

Wie im letzten Abschnitt beschrieben, sind in den klassischen Szenarien des automatischen Planens die Aktionen dimensionslos und passieren unmittelbar. Um reale Umgebungen besser abzubilden, kann die Zeit als Ressource eingeführt werden. Dies ermöglicht Ereignisse nach ihrem zeitlichen Auftreten zu beurteilen. Ein typisches Hilfsmittel dafür stellt die Intervallalgebra von Allen dar. Diese wird im Folgenden beschrieben. Als erstes wird die Algebra allgemein betrachtet und da die vollständige Algebra NP-vollständig ist, werden anschließend effizient lösbare Unterklassen vorgestellt.

#### 2.5.1. Intervallalgebra

Die Stärke der Intervallalgebra ist, dass zeitlich in Beziehung stehende Ereignisse geordnet werden können ohne genaue Zeitpunkte zu kennen. Ereignisse werden als Zeitintervalle betrachtet, die

---

<sup>11</sup>Distributed Constraint Satisfaction Problem

<sup>12</sup>Hierarchical Task Network

Relation	Symbol	Symbol for Inverse	Pictoral Example
$X$ before $Y$	<	>	XXX YYY
$X$ equal $Y$	=	=	XXX YYY
$X$ meets $Y$	m	mi	XXXXYY
$X$ overlaps $Y$	o	oi	XXX YYY
$X$ during $Y$	d	di	XXX YYYYYY
$X$ starts $Y$	s	si	XXX YYYYY
$X$ finishes $Y$	f	fi	XXX YYYYY

Abbildung 2.9.: Beziehungen von Zeitintervallen aus Allen (1983). Die Relationen < und > werden in der vorliegenden Arbeit mit  $b$  (before) und  $a$  (after) bezeichnet.

in einem relativen Verhältnis zueinander stehen. Nicht explizit definierte Relationen können mit Hilfe der Intervallalgebra geschlussfolgert werden. Zum Beispiel findet Ereignis A vor Ereignis B statt, aber C ereignet sich während B. Aus diesen Zusammenhängen kann geschlossen werden, dass Ereignis A auch vor C stattfindet.

Allen (Allen, 1983, S. 835) beschreibt eine Algebra, die auf Zeitintervallen basiert. Ein Zeitintervall stellt ein Ereignis mit einer zeitlichen Ausdehnung dar. Wie groß diese Ausdehnung in der Wirklichkeit ist, spielt keine Rolle. Es können die Prozesse eines Computers genauso dargestellt werden, wie Abläufe in einer Fabrik. Definiert werden Zeitintervalle als  $t$  das aus zwei Zeitpunkten  $t^-$  und  $t^+$  besteht, mit der Beziehung  $t^- < t^+$ . Das besondere an der Intervallalgebra ist, dass sie Verhältnisse zwischen Intervallen angibt, ohne feste Zeitpunkte zu benötigen.

So kann z. B. entschieden werden, ob zwei beliebige Intervalle in einem Netz von Constraints nacheinander ablaufen oder sich überlappen. Allen zeigt 13 mögliche Beziehungen zwischen zwei Intervallen (siehe Abbildung 2.9).

Davon ausgehend können die Relationen zwischen zwei Intervallen als Relationen zwischen ihren jeweiligen Endpunkten dargestellt werden. Dies zeigt Tabelle 2.1.

Allens *during*-Relation wird von ihm in drei Relationen untergliedert und zwar in *during*, *starts* und *finished* (d, s, f). Inklusiv der drei so genannten Containment-Relationen (di, si, fi) kommt Allen auf 13 Intervallrelationen.

Tabelle 2.1.: Intervall-Relationen und ihre Endpunkt-Relationen (Allen, 1983, S. 8, Figure 1).

Interval Relation	Equivalent Relation on Endpoints
$t < s$	$t+ < s-$
$t = s$	$(t- = s-) \& (t+ = s+)$
$t \text{ overlaps } s$	$(t- < s-) \& (t+ > s-) \& (t+ < s+)$
$t \text{ meets } s$	$t+ = s-$
$t \text{ during } s$	$((t- > s-) \& (t+ = < s+)) \text{ or } ((t- > = s-) \& (t+ < s+))$

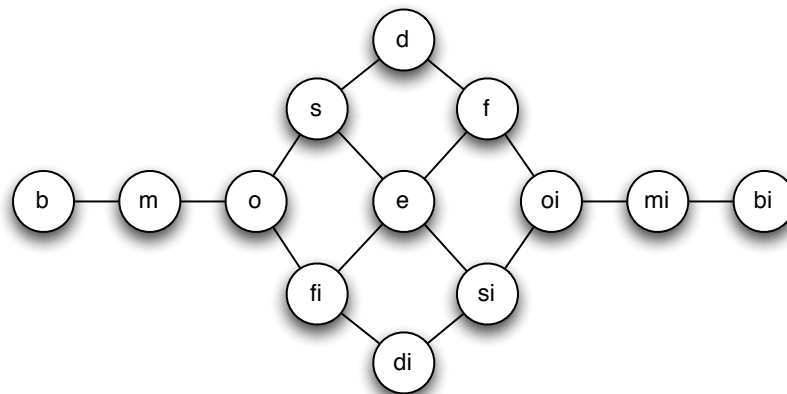


Abbildung 2.10.: Graph für die Konvexitätsbedingung in der Intervallalgebra (Nau u. a., 2004, S. 301) ( $f'$  usw. aus der original Abbildung sind hier zu  $fi$  usw. angepasst)

### 2.5.2. Effizient lösbare Unterklassen von Allens Intervallalgebra

Da die vollständige Intervallalgebra von Allen  $\mathcal{IA}$  zwar entscheidbar, aber nicht effizient lösbar ist (NP-vollständig), wurden inzwischen mehrere Subklassen dieser Algebra gebildet, die in polynomieller Zeit gelöst werden können. In Krokhin u. a. (2003) werden alle effizient lösbaren Unterklassen der Intervallalgebra aufgeführt, und es wird bewiesen, dass die 18 vorgestellten Klassen alle existierenden Unterklassen, die effizient gelöst werden können, darstellen. Die Klasse  $\mathcal{H}$  aus Nebel und Bürckert (1995) stellt mit ihren 868 Kombinationsmöglichkeiten die größte dieser Unterklassen dar (Allens Intervallalgebra besitzt  $2^{13} = 8192$  Kombinationsmöglichkeiten).

In Nau u. a. (2004)<sup>13</sup> wird eine berechenbare Unterklasse  $\mathcal{IA}_c$  von  $\mathcal{IA}$  durch die Konvexitätsbedingung hergeleitet. Die Konvexitätsbedingung kann mit dem in Abbildung 2.10 gezeigten Graph nachvollzogen werden. Eine Bedingung ist konvex, wenn neben den zwei Relationen  $i$

<sup>13</sup>S. 301

und  $j$  auch alle Relationen enthalten sind, die auf dem kürzesten Pfad zwischen  $i$  und  $j$  liegen. Zum Beispiel ist  $\{m, o, s, d\}$  konvex, während  $\{o, s, d, f\}$  nicht konvex ist, da zwischen  $o$  und  $f$  noch die Pfade  $o, s, e, f$  und  $o, fi, e, f$  existieren und  $e$  und  $fi$  in der Bedingung fehlen. Die Klasse  $\mathcal{IA}_c$  aus [Nau u. a. \(2004\)](#) besitzt 82 Bedingungen. Die Entscheidung für eine Unterklasse hängt von der benötigten Ausdrucksstärke ab.

### 2.5.3. Fazit zur Zeitalgebra

Mit Hilfe der Intervallalgebra ist es möglich qualitative Aussagen zu zeitlichen Verhältnissen zwischen Ereignissen zu treffen und die Konsistenz dieser Aussagen zu überprüfen. Die von Allen eingeführte vollständige Algebra hat den Nachteil, dass sie NP-vollständig ist. Ein Reasoning ist damit nicht effizient umzusetzen.

Es gibt eingeschränkte Unterklassen von Allens Intervallalgebra mit unterschiedlicher Ausdrucksstärke, die effizient lösbar sind. Je nachdem welche Ausdrucksstärke benötigt wird und wieviel Ressourcen für das Reasoning aufgewandt werden sollen, kann eine dieser Unterklassen ausgewählt werden. Über die Konvexitätsbedingung kann zudem ein in sich konsistenter Satz an Relationen z. B. für Testzwecke zusammen gestellt werden.

## 3. Analyse

In den Grundlagen sind Smart-Home-Projekte aufgeführt, die eher die generellen technischen Hürden, die lose Kopplung und das Auffinden von Services behandelt haben. Nur das CoFriend-Projekt hat sich mit der Interpretation von Verhalten beschäftigt und versucht Abläufe und Aktivitäten zu erkennen. Im Gegensatz zum CoFriend-Projekt behandelt die vorliegende Arbeit die Vorhersage von Abläufen und Aktivitäten, auf Grundlage dessen, was vorher ein Interpretierer erkannt hat. Aus der Vorhersage sollen Anweisungen resultieren, die dafür sorgen, dass sich die Smart-Home-Umgebung auf die Situation einstellt.

Wie aus dem Grundlagenkapitel über das automatische Planen bekannt, besteht beim automatischen Planen das Problem, dass Aktionspläne ein zu starres Konstrukt und oft zu unflexibel für dynamische Umgebungen sind. In Kapitel 3 wird für diese Arbeit der Ansatz einer dynamischen Planung von Zielen statt Aktionen gewählt. Hierbei werden Ziele als in der Zukunft liegende angestrebte partielle Zustände des Gesamtsystems definiert.

Dies geschieht mit der Annahme, dass durch die Möglichkeit der späten Bindung (least commitment) eine dynamischere Reaktion auf Kontextänderungen erfolgen kann. Damit wäre der Nachteil des klassischen automatischen Planens vermieden. Der Nachteil besteht darin, Pläne verwerfen zu müssen, wenn während der Ausführung eines Planes Kontextänderungen eintreten, die den bisherigen Plan obsolet werden lassen. Indirekt folgt dies der Idee aus [de Weerd u. a. \(2005\)](#) eines *Distributed Continual Plannings*.

Im Folgenden werden die in der Zielsetzung genannten Szenarien systematisch beschrieben und analysiert (Abschnitt 3.1), anschließend werden die Erkenntnisse der Analyse ausgearbeitet (Abschnitt 3.2 - Abschnitt 3.3). Nach dem Fazit (Abschnitt 3.4) wird erläutert, was auf Grundlage dieser Erkenntnisse erreicht werden kann (Abschnitt 3.5). Am Ende der Analyse werden Anforderungen an das Design des Prototyps gestellt (Abschnitt 3.6), welches im darauf folgenden Kapitel (4) vorgestellt wird.

### 3.1. Zwei Userstories

In diesem Abschnitt werden zwei Userstories ausgearbeitet, die beispielhaft das Context-Aware-Computing im Living Place charakterisieren sollen. Aufbauend auf dem Wecker 2.0 aus [Ellenberg](#)

(2010) wird als erstes eine Story für einen Tagesbeginn vorgestellt, anschließend der Besuch von Freunden. Danach werden diese Userstories analysiert.

Die Userstories sind eine Hommage an Mark Weiser und orientieren sich an der Beschreibung des Tagesablaufs der Protagonistin Sal aus Weiser (1991). Aus diesem Grund wird in den Userstories die intelligente Wohnung Living Place von Sal junior bewohnt.

#### 3.1.1. Userstory 1: Tagesbeginn

Userstory 1 beschreibt einen Morgen in Sal jr.'s Wohnung:

1. Sal jr. möchte gern mit Licht geweckt werden. Weil die Sonne scheint, lässt die Wohnung daher die Gardinen zurückfahren, um sie mit dem Tageslicht zu wecken.
2. Sal jr. mag es warm in der Wohnung, wenn es aufsteht. Da es draußen kalt ist, wird die Heizung von der Wohnung rechtzeitig hochgeschaltet.
3. Nach dem Aufstehen geht Sal jr. zunächst immer duschen.
4. Während Sal jr. duscht, veranlasst die Wohnung Kaffee zu kochen. Wenn Sal jr. aus der Dusche kommt, riecht sie den Kaffee.
5. Sie trinkt zum Frühstück den Kaffee. Unterdessen meldet ihr die Wohnung, dass auf dem Reader, der auf dem Tisch liegt, die Unterlagen für Sal jr.'s erstes Meeting geladen wurden.
6. Sal jr. ignoriert den Reader und frühstückt.
7. Die Wohnung meldet Sal jr., dass sie die U-Bahn um 8.30 Uhr nehmen müsse, um rechtzeitig beim Meeting zu sein, d.h. sie müsste um 8.20 Uhr aus dem Haus.
8. Da Sal jr. anscheinend die Unterlagen nicht anschauen möchte und noch Zeit ist sie das Haus verlassen muss, erinnert die Wohnung sie, dass sie die Zeit nutzen könnte, um beim Reisebüro eine Buchungsanfrage für den Urlaub zu stellen.

#### 3.1.2. Userstory 2: Besuch von Freunden

Die zweite Userstory beschreibt einen möglichen Ablauf für die Situation, dass Freunde zu Besuch eingeladen sind:

1. Sal jr. trägt in ihrem Terminkalender für den nächsten Abend drei Vornamen ein. Die drei Personen werden zu einem Essen in ihre Wohnung eingeladen.

### 3. Analyse

---

2. Die Wohnung findet im Adressbuch mehrere Personen mit diesen Vornamen, teils Geschäftspartner, teils Freunde.
3. Die Wohnung erfragt bei Sal jr., ob es sich um einen privaten oder geschäftlichen Termin handelt.
4. Sal jr. informiert die Wohnung, dass es sich um einen privaten Termin handelt.
5. (optional) Die Wohnung ordnet im Adressbuch die drei Personen der Kategorie Freunde zu.
6. Die Wohnung ermittelt Gerichte, die Sal jr. bei bisherigen Treffen gerne gekocht hat und fragt Sal jr., ob sie eines dieser Gerichte auswählen möchte.
7. Sal jr. wird die Liste präsentiert.
8. Sal jr. wählt ein Gericht aus.
9. Die Wohnung ermittelt was an benötigten Zutaten für das Gericht in den Vorräten fehlt und stellt eine Einkaufsliste zusammen.
10. Sobald Sal jr. in der Nähe eines Lebensmittelgeschäftes befindet, erinnert die Wohnung sie, dass sie noch einkaufen müsste.

#### 3.1.3. Analyse der Userstories

Was in den Userstories verallgemeinert als „die Wohnung“ bezeichnet wird, teilt sich in verschiedene Subsysteme auf. Diese Subsysteme werden hier allgemein als Agenten und Dienste bezeichnet. Agenten<sup>1</sup> stellen in diesem Kontext proaktive autonome Einheiten dar. Dienste sind passive Schnittstellen, die z.B. von Agenten genutzt werden können.

Beispielhaft sollen am Punkt eins der ersten Userstory Implikationen aufgezeigt werden, die sich durch die für beide Userstories aufgeführten Punkte eins bis acht bzw. eins bis neun jeweils ergeben:

1. Es besteht das **Ziel**, dass Sal jr. zu einer bestimmten Uhrzeit wach ist.
2. Es sind Informationen darüber verfügbar in welchem Zimmer sich Sal jr. aufhält und sogar über ihren Schlaf-/Wachzustand.
3. Es ist die Information verfügbar, dass Sal jr. mit Licht geweckt werden möchte.

---

<sup>1</sup>Eine Definition des Agenten-Begriffs findet sich z. B. in [Tennstedt \(2007\)](#) S. 15ff.



### 3. Analyse

---

4. Es ist das Wissen darüber vorhanden, wie sich das Wetter auf die Lichtsituation auswirkt.
5. Das aktuelle Wetter ist bekannt.
6. Es werden eine oder mehrere **Aktionen** ausgeführt, die zur Folge hat/haben, dass die Vorhänge sich öffnen, um Tageslicht ins aktuelle Aufenthaltszimmer scheinen zu lassen.

Wie in der Zielsetzung auf Seite 2 beschrieben, wird untersucht, wie ein System aufzubauen ist, das anstatt Anweisungen Ziele plant. So besteht in diesem Beispiel am Anfang ein Ziel. In diesem Fall, dass Sal jr. zu einer bestimmten Uhrzeit wach sein soll. Daraus ergeben sich die Fragen über den aktuellen Kontext der Wohnung und die Entscheidung, wie das Ziel zu erfüllen ist. Das mündet am Ende in die oben beschriebenen Aktionen zum wecken mit Licht.

Im obigen Beispiel steht am Anfang ein Ziel und am Ende eine oder mehrere Aktionen. Die Informationen dazwischen werden als gegeben angenommen. Diese müssen eigentlich jedoch zuerst beschafft und ausgewertet werden. Um dies zu verdeutlichen wird ein weiteres Beispiel, diesmal aus der Userstory 2, herangezogen. Hier wird die Auswahl eines Gerichtes in Einzelschritten dargestellt:

1. Es besteht das **Ziel** ein Gericht für das Treffen ausgewählt zu haben.
2. Es werden in einer **Aktion** Informationen über bisher präferierte Gerichte von Sal jr. zusammengetragen.
3. Es wird in einer **Aktion** Sal jr. die Auswahlliste präsentiert.
4. Es folgt die **Aktion**, in der Sal jr. ein Gericht auswählt.

Das in dieser Arbeit zu untersuchende System soll das Ziel „ein Gericht für das Treffen ist ausgewählt“ planen. Die Wohnung als Gesamtsystem führt anschließend Aktionen aus, um das gesetzte Ziel zu erfüllen. Die Aktionen des Gesamtsystems entstehen implizit durch die Setzung eines Ziels. Damit können selbst komplexe Abläufe durch ein einfaches Ziel zum Ausdruck gebracht werden.

#### **Der Bewohner „in the loop“**

Ein weiterer Aspekt ist die Interaktion mit dem Benutzer. In Userstory 2 heißt es im sechsten und siebten Punkt, dass die Wohnung Gerichte ermittelt und Sal jr. eine Liste von Gerichten als Vorschlag zur Auswahl präsentiert. Anschließend wählt Sal jr. ein Gericht aus. Mit dem Beispiel aus Userstory 2 wird gezeigt, wie die Wohnung automatisch mit mehreren Aktionen alle benötigten Informationen ermittelt und das Ziel erfüllt.

Bei Entscheidungen, die die Wohnung nicht allein treffen kann oder soll, werden Rückfragen an den Bewohner gestellt und dieser damit „in the loop“ gehalten. Dies erhöht die Transparenz der Arbeit des Systems. Der Bewohner zeichnet sich als der Verantwortliche für die Ziele aus.

#### **Userstory und Szenario**

Die beiden Userstories bestehen aus mehreren Zielen, die zusammengenommen jeweils ein bestimmtes Szenario abbilden. Diese Ziele bauen teilweise aufeinander auf. Zum Beispiel wäre es in Userstory 2 nicht zielführend erst die Einkaufsliste zusammenzustellen und dann das Gericht zu ermitteln. Damit kann festgestellt werden, dass die Ziele eines Szenarios teilweise oder vollständig geordnet und zudem teilweise oder vollständig voneinander abhängig sind.

Die Userstories spiegeln die Gewohnheiten von Sal jr. wider. Sie duscht nach dem Aufstehen. Wenn Freunde zu Besuch kommen, wird gekocht. Der Morgen kann bei einem anderen Bewohner natürlich anders ablaufen. Eine andere Person würde vielleicht zuerst Frühstück, dann Duschen und Zähneputzen oder erst Duschen, dann Frühstück und dann Zähneputzen, also zweimal ins Bad gehen. Eine Zusammenstellung solcher Art Gewohnheiten wurde anfangs in den zwei Userstories beschrieben. Im Folgenden wird die technische Abbildung der Userstory als Szenario bezeichnet. Im Tagesbeginn-Szenario ist es für die Wohnung irrelevant, wann der Bewohner seine Zähne putzt, wichtig ist, wann er frühstückt, damit die Wohnung weiß, wann der Kaffee fertig sein soll.

Solange die Abläufe so stattfinden wie in der Userstory angegeben, handelt es sich um das korrespondierende Szenario. Ändert Sal jr. ihr Verhalten, zum Beispiel zieht sie sich beim Tagesbeginn nicht an, so befindet sie sich anscheinend nicht länger in dem Tagesbeginn-Szenario, das zu der Beschreibung in Userstory 1 passt. Vielleicht ist Sal jr. krank und bleibt zuhause. Die Wohnung kennt im Idealfall für diese Situation das passende Szenario.

## **3.2. Charakterisierung der Ziele**

In diesem Abschnitt werden die Ziele genauer betrachtet und nach bestimmten Eigenschaften unterschieden. Im Gegensatz zu einem Aktionsplan im automatischen Planen, der aufzeigt, wann was ausgeführt werden soll, geben Ziele an, was am Ende erreicht werden soll. Die Aufgabe des Planers verschiedene Aktionen aneinanderzuketten, um ein Ziel zu erreichen, wird beim Planen von Zielen durch die Arbeit der Agenten im System ersetzt. Damit stehen Ziele den Aktionen beim automatischen Planen gegenüber. Sie sind der feine Unterschied zwischen Taten und Wünschen.

#### 3.2.1. Herleitung der Ziele aus den Userstories

In Abschnitt 3.1 wurden exemplarisch Userstories gezeigt, die das zu erarbeitende System abdecken soll. Der Zielbegriff, wie er in dieser Arbeit verwendet wird, wurde bereits verdeutlicht. Für die folgenden Abschnitte ist es notwendig die Userstories durch eine Zusammenstellung von Zielen abzubilden. In diesem Abschnitt wird dieser Schritt getan.

In Unterabschnitt 3.1.3 heißt es, dass in Userstory 1 als erstes das Ziel besteht, dass Sal jr. wach ist. Analog dazu werden die andere Punkte von der Userstory in Zielen ausgedrückt. Daraus ergibt sich folgende Liste von Zielen:

1. Sal jr. ist um 7.00 Uhr wach.
2. Wohnraumtemperatur beträgt ab 7.00 Uhr 20,5°C.
3. Sal jr. hat geduscht.
4. Sal jr. ist angezogen.
5. Kaffee ist gekocht.
6. Sal jr. hat gefrühstückt.
7. Unterlagen für Meeting sind bereitgestellt.
8. Sal jr. ist über Tagesablauf informiert.
9. Sal jr. hat Wohnung spätestens um 8.20 Uhr verlassen.
10. Eingeschoben: Urlaub ist gebucht.

Die Ziele sind eine geradlinige Abbildung der Userstory 1 aus Abschnitt 3.1.1. Hingegen ergibt sich für die Userstory 2, dem Besuch von Freunden, eine komplexere Abbildung.

1. Ein Termin zu einem privaten Treffen existiert.
  - a) (Hilfsziel) Typ des Treffens ist zugeordnet.
2. Ein Gericht für das Treffen ist ausgewählt.
  - a) (Hilfsziel) Auswahl von Gerichten ist erstellt.
3. Es sind genügend Lebensmittel vorhanden.
  - a) (Hilfsziel) Einkaufsliste ist erstellt.




- (1)  Zielerfüllung beginnt zum Anfang des Intervalls.
- (2)  Zielerfüllung ist zum Ende des Intervalls terminiert.
- (3)  Zielerfüllung ist spätestens zum Ende des Intervalls terminiert.

Abbildung 3.1.: Zieltypen unterschieden nach Zeitpunkt der Zielerfüllung.

b) (Hilfsziel) Einkauf ist erledigt.

4. Der Besuch ist eingetroffen.

Für diese Userstory können Hilfsziele angewandt werden. Diese Hilfsziele sind im automatischen Planen mit Hilfsplänen vergleichbar. Sie werden eingesetzt, wenn das übergeordnete Ziel ohne sie nicht erfüllt werden kann. Wird zum Beispiel ein privater Termin als solcher gekennzeichnet im Kalender eingetragen, bzw. kann das System anhand der Teilnehmer schließen, dass es sich um einen privaten Termin handelt, so ist das dazugehörige Hilfsziel nicht notwendig, denn das eigentliche Ziel ist bereits erfüllt. Kann das System andererseits den Termin nicht zuordnen, so ist es für Userstory 2 notwendig zuerst herauszufinden, welcher Art das Treffen ist. Genauso verhält es sich beim dritten Hauptziel der Userstory 2. Wenn die für ein Gericht benötigten Lebensmittel vorhanden sind, braucht keine Einkaufsliste erstellt werden und es ist nicht nötig einzukaufen. Die Hilfsziele für dritte Hauptziel brauchen in dem Fall nicht geplant werden.

#### 3.2.2. Zieltypen

Aus den im vorherigen Abschnitt aufgezeigten Zielen sollen nun unterschiedliche Zieltypen herausgearbeitet werden.

Abbildung 3.1 zeigt die symbolische Darstellung der im Folgenden herausgearbeiteten drei Zieltypen.

Ein Ziel wie „Ein Termin zu einem privaten Treffen existiert.“ impliziert, dass der Termin an einem bestimmten Zeitpunkt eingetragen wird und ab da existiert, bis der Termin tatsächlich eintrifft. Diese Tatsache ist wichtig, damit andere Ziele, die mit dem Termin zusammenhängen, geplant werden können. Würde der Termin gelöscht, würden andere mit dem Termin verbundene Ziele hinfällig werden. Dieser Zieltyp ist am Anfang eines Zeitintervalls erfüllt und bleibt es bis zum Ende des Intervalls.

Zu diesem Typ gehören die Ziele:

Userstory 1: 8

Userstory 2: 1, 2a, 3a

In Abbildung 3.1 steht das erste Symbol für diesen Zieltyp. Charakteristisch für ihn ist, dass die Erfüllung des Ziels *ab* einem bestimmten Zeitpunkt gewährleistet ist und dies eine gewisse Zeit lang bleibt.

Ein Ziel wie „Sal jr. ist um 7.00 Uhr wach.“ beinhaltet, dass Sal jr. geweckt werden muss. Dieser Weckvorgang dauert eine Zeit lang, ist dieser aber abgeschlossen, ist Sal jr. wach. Dieser Zieltyp ist am Ende eines Zeitintervalls erfüllt.

Zu diesem Typ zählen die Ziele:

Userstory 1: 1, 2, 3, 4, 5

Userstory 2: 1a, 2, 3

Abbildung 3.1 zeigt unter (2) das Symbol dieses Zieltyps. Merkmal dieses Zieltyps ist es, dass die Erfüllung eines Ziels genau *zu* einem bestimmten Zeitpunkt gewährleistet ist. Die Zielerfüllung besitzt in der Regel einen gewissen Vorlauf, doch zu dem bestimmten Zeitpunkt muss sie abgeschlossen sein.

Ein Ziel der Art „Bewohner ist über Tagesablauf informiert.“ impliziert, dass der Bewohner spätestens zu einem bestimmten Zeitpunkt informiert werden muss. Wünschenswert ist es natürlich, dass es vor dem spätmöglichen Zeitpunkt geschieht.

Zu diesem Typ gehören die Ziele:

Userstory 1: 6, 7

Userstory 2: 3b

In Abbildung 3.1 kennzeichnet das dritte Symbol Ziele, deren Erfüllung *bis spätestens* zu einem bestimmten Zeitpunkt gewährleistet ist, die Ziele können aber auch schon vorher erfüllt werden.

Die Ziele der hier vorgestellten Userstories lassen sich vollständig und disjunkt in diese drei Typen aufteilen. Die Typenunterteilung hat nicht den Anspruch der Vollständigkeit. Bei Betrachtung von weiteren Userstories lassen sich u.U. weitere Typen definieren. Für die Zielsetzung dieser Arbeit würde eine höhere Typenanzahl keine grundlegenden weiteren Erkenntnisse liefern.

#### 3.2.3. Zeitrelation der Ziele untereinander

Unabhängig davon welchen Typs ein Ziel ist, besitzen Ziele eine zeitliche Ausdehnung und können unter Anwendung von Allens Intervallalgebra (Allen, 1983, S. 835ff), als Zeitintervalle betrachtet werden, die zueinander in bestimmten Relationen stehen. Wie Handlungen und Ereignisse mit Hilfe der Intervallalgebra in Relation gebracht werden, wird in Allen (1983)<sup>2</sup> und in Nau u. a. (2004)<sup>3</sup> erklärt. Die Kürzel der Intervallrelationen sind in Abbildung 2.9 aufgeführt.

---

<sup>2</sup>S. 837ff

<sup>3</sup>S. 293ff

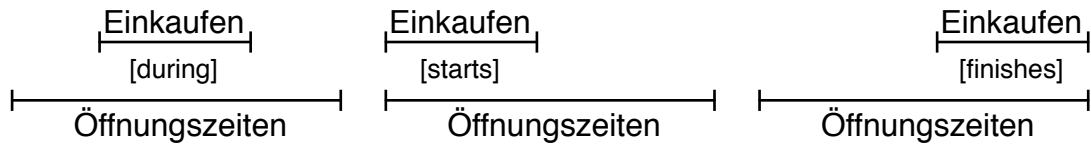


Abbildung 3.2.: Grafische Darstellung der Intervallrelation Einkaufen zu Öffnungszeiten

Als Beispiel, wie zwei Intervalle in Relation gebracht werden können, wird ein Intervall mit dem Namen *Einkaufen* herangezogen. Als Bezug kann die Ladenöffnungszeit genommen werden. Ein Einkauf muss erledigt sein, bevor die Geschäfte schließen. Naiv betrachtet führt dies zu:

$$\text{Einkaufen} \xrightarrow{\{f\}} \text{Öffnungszeiten}$$

Der Einkauf endet zum Ende der Ladenöffnungszeiten. Doch ein Einkauf kann zu Beginn der Ladenöffnungszeit anfangen, zum Ende der Ladenöffnungszeiten beendet werden und jederzeit während der Ladenöffnungszeiten stattfinden. Daher handelt es sich streng genommen um folgende Beziehung:

$$\text{Einkaufen} \xrightarrow{\{s,d,f\}} \text{Öffnungszeiten}$$

Dieser Sachverhalt wird in Abbildung 3.2 grafisch dargestellt. Die Relationen sind *ODER* verknüpft, d. h. das Einkaufen kann zur Ladenöffnung beginnen (*s*) oder währenddessen stattfinden (*d*), sowie genau zur Ladenschließung beendet werden (*f*).

Um zu zeigen, wie Ziele mit Hilfe der Intervallalgebra in Relation gebracht werden können, wird beispielhaft das Ziel „Ein Termin zu einem privaten Treffen existiert.“ herangezogen. Es wird mit dem Ziel „Der Besuch ist eingetroffen“ in Relation gesetzt.

$$\text{privater Termin existiert} \xrightarrow{\{o,m,b\}} \text{Besuch ist eingetroffen}$$

Der Termin wurde vielleicht zwei Tage vor dem Zeitpunkt des Treffens im Kalender eingetragen. Der Besuch trifft zu früh (*overlaps*), pünktlich (*meets*) oder zu spät (*before*) ein oder kommt gar nicht, was hier nicht berücksichtigt wird.

Als ein weiteres Beispiel wird das Ziel „Ein Termin zu einem privaten Treffen existiert.“ mit dem Ziel „Ein Essen ist gekocht.“ in Relation gebracht.

$$\text{privater Termin existiert} \xrightarrow{\{di,fi\}} \text{Essen ist gekocht}$$

Das Essen soll zum Zeitpunkt des Besuchs fertig sein, daher besteht die Relation, dass das Essen gekocht ist, wenn das andere Zeitintervall durch das Eintreten des Termins abgeschlossen wird (*finishes inverse*). Genau genommen kann das Essen auch früher fertig sein, deswegen besteht auch die Relation, dass das Kochen während (*during inverse*) des Zeitraums passiert, in dem der Termin existiert. Allerdings ist letzteres zu ungenau, da der Termin wie oben erwähnt worden ist zwei Tage vor dem Zeitpunkt des Treffens eingetragen wurde. Wenn festgestellt wird, dass einen halben Tag vor dem Treffen ein Essen gekocht wird, ist es möglich, dass dieses Essen nicht das ist, das für den Besuch gedacht ist.

Zum Einplanen eines Ziels ist es nicht notwendig Start- und Endzeiten seitens des Planers anzugeben. Die Instanzen, die die Ziele erfüllen, besitzen das nötige Wissen um konkrete Zeiten. Konkrete zeitliche Grenzen werden also durch unmittelbares Wissen gesetzt. Wenn in einem Terminkalender etwas vermerkt wurde oder ein Agent eine Grenze festgestellt hat (z.B. die Öffnungszeiten), dann können diese in den Zielen hinterlegt werden. Durch die Relationen zwischen den Zielen, sind die abhängigen Ziele automatisch zeitlich positioniert.

#### 3.2.4. Voraussetzung zum Erreichen eines Ziels

Dieser Abschnitt beschäftigt sich mit der Frage, welche Voraussetzungen gelten müssen, damit ein Ziel aus Sicht des Systems als erfüllt bezeichnet werden kann.

Dazu können mehrere Herangehensweisen gewählt werden. Diese sollen hier miteinander verglichen werden.

Bei dieser Betrachtung wird angenommen, dass prinzipiell jede Systemkomponente Kenntnis über alle bestehende Ziele des Systems haben kann und Konsens darüber besteht, welche Ziele noch zu erfüllen sind. Diese Annahme wird mit dem Begriff Blackboard umschrieben, der Begriff soll aber keine architektonische Festlegung sein.

Ein Ziel gilt für das System als erfüllt,

1. wenn der beschriebene Zielkontext sich eingestellt hat oder,
2. wenn der verantwortliche Agent meldet, dass er das Ziel erreicht hat oder,
3. wenn es sich nicht mehr auf dem Blackboard befindet.

Im ersten Fall würde ein zentral koordinierender Agent die Erfüllung eines Ziels überprüfen. Dieser Agent würde kontinuierlich den aktuellen Kontext mit dem gewünschten Zielzustand abgleichen. Ist der Zielzustand eingetreten, markiert er das jeweilige Ziel als erfüllt oder löscht es vom Blackboard. Wird das erfüllte Ziel markiert, würde ein für das Reinigen des Blackboards zuständiger Agent das Ziel vom Blackboard löschen.

Im zweiten Fall hätte der für das Ziel verantwortliche Agent Wissen über den speziellen Bereich des Gesamtkontext, der mit dem Ziel zusammenhängt. Sobald der Agent feststellt, dass sich der partielle Kontext wie gewünscht eingestellt hat, würde er das Ziel als erfüllt kennzeichnen. Wie im Fall zuvor würde ein Agent, der mit der Reinigung des Blackboards betraut ist, das Ziel anschließend löschen.

Im dritten Fall würde der Agent, der sich einem Ziel annimmt, dieses vom Blackboard löschen. Das System nimmt dadurch an, dass das Ziel erfüllt ist, obwohl mit dem Löschen des Ziels vom Blackboard nur ein Agent gefunden wurde, der sich mit der Erfüllung des Ziels befasst.

Aus menschlicher Sicht ist die erste Variante intuitiv die Sicherste. Nur wenn einer übergeordneten Instanz wirklich klar ist, dass das Ziel erreicht ist, wird es auch gelöscht. In der zweiten Variante wird dem ausführenden Agenten zwar mehr Einfluss gewährt, doch auch hier muss eine andere Instanz die Entscheidung endgültig machen.

Die dritte Variante hingegen arbeitet nach dem Prinzip des Vertrauens. Wer sich um ein Ziel kümmert, übernimmt die volle Verantwortung dafür.

Anzunehmen ist, dass das System generell aus vertrauenswürdigen, also zuverlässigen, Agenten besteht. Gemeint ist, dass keine destruktiven Agenten existieren, die Ziele löschen ohne sie zu bearbeiten, was einen Fehler repräsentieren würde.

Aus diesem Grund soll die dritte Variante, ein Agent löscht das Ziel vom Blackboard, sobald er sich der Erfüllung eines Ziels annimmt, als Grundlage für das hier erarbeitete Design gewählt werden. Der Vorteil liegt in der Einfachheit des Konzept, denn es ist kein eigener Agent zur Bereinigung des Blackboards notwendig oder eine andere Instanz, die dies überprüft.

#### **Was passiert, wenn ein Agent ein Ziel nicht erfüllt, obwohl er annimmt er hätte es erfüllt?**

Das ist ein generelles Problem des automatischen Planens und Handelns und wird als Ramification Problem (Nau u. a., 2004, S. 343) bezeichnet. Dieses Problem kann sich auf unterschiedliche Weisen zeigen. Zum einen können die Handlungen, die ein System ausführt, Nebeneffekte haben, die nicht mit modelliert sind. Diese Nebeneffekte können zur Folge haben, dass andere Handlungen des Systems behindert werden und damit nicht erfolgreich sind, obwohl das System das Gegenteil annimmt. Zum anderen können störende Einflüsse von außen verhindern, dass Handlungen des Systems den gewünschten Effekt haben. Treten solche Störungen auf, muss das System über Hilfs- bzw. Repairpläne verfügen, um die Voraussetzungen zu schaffen, die nötig sind, um die eigentlichen Ziele zu erreichen.



#### Was passiert, wenn ein Ziel unerfüllbar ist?

Wie bei der vorangegangenen Frage ist auch das Erkennen von unerfüllbaren Zielen ein generelles Problem von automatisch planender und handelnder Systeme.

Die Frage, wie eine Prognose erstellt werden kann, ob ein Ziel unerfüllbar ist, wird im Rahmen der vorliegenden Arbeit nicht behandelt. Dieser Frage sollte sich eine weiterführende Arbeit annehmen.

Es gibt zwei Arten der Unerfüllbarkeit. Ein Ziel ist zunächst unerfüllbar, es sind allerdings Hilfsziele vorhanden, die die Bedingung für die Erfüllbarkeit herstellen können. Ein Ziel ist unerfüllbar und das System hat keine Handlungsmöglichkeit, die Voraussetzungen für eine Erfüllbarkeit herzustellen.

Das System muss mit oder ohne einer solchen Erkennung robust genug sein, um seine Gesamtfunktionalität auch bei derartigen Problemen zu erhalten.

#### 3.2.5. Definition der Zieldeklaration

Ziele werden als folgendes Tupel dargestellt:

$$Z = (Was, Wo, Wer, Wann, Wie lang, RelationZu) \quad (3.1)$$

Diese Eigenschaften haben folgende Bedeutung:

- Was** Zielstatus z. B. Person ist wach.
- Wo** Auf einen Ort beschränkt? Z. B. Wohnung.
- Wer** Bezogen auf eine Person oder Ressource? Z. B. Sal jr..
- Wann** Zieltyp und Zeit z. B. Typ 2, 07.00 Uhr.
- Wie lang** Zeitliche Ausdehnung des Zeitintervalls, um das Ziel zu erreichen z. B. 15min.
- RelationZu** Gibt es eine relative temporale Abhängigkeit zu anderen Zielen? Z. B. *before* Ziel<sub>i</sub>.

### 3.3. Herleiten des Szenarios

In **Schank und Abelson (1977)** werden soziale Situationen als Handlungsskripte, quasi als Planfragmente, ausgearbeitet. Die von Schank und Abelson beschriebene Restaurant-Situation ist ein Beispiel dafür: Der Gast betritt das Restaurant, bekommt vom Kellner einen Platz zugewiesen, die Karte gereicht und bestellt etwas zu trinken. Anschließend bekommt der Gast sein Getränk und nennt dem Kellner das gewählte Gericht. Nach einer Weile bringt der Kellner das Essen. Hat

### 3. Analyse

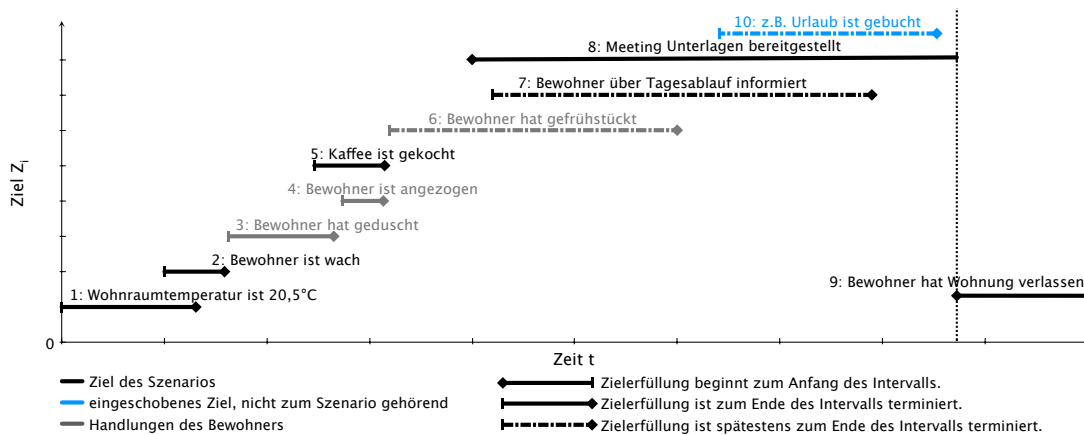


Abbildung 3.3.: Erste Skizze des Szenarios für Userstory 1 von Seite 22.

der Gast getrunken und gegessen und hat keinen weiteren Wunsch, bekommt er vom Kellner die Rechnung. Daraufhin übergibt der Gast dem Kellner Geld. Zum Ende verlässt der Gast das Restaurant. Die Situation ist damit abgeschlossen.

Im Gegensatz zu den Handlungsskripten, die im Prinzip Abfolgen von Aktionen darstellen, beschreiben Szenarios die erwarteten Ergebnisse aus vorangegangenen Aktionen. Die erwarteten Ergebnisse werden mit den vorher in dieser Arbeit eingeführten Zielen ausgedrückt. Diese Ziele werden zu Skripten zusammengefasst und im Folgenden als *Szenario* bezeichnet werden.

Die Szenarios stellen einen vordefinierten Zielgraphen dar, der dem System als Schablone dienen soll, um die Szenarios auf bestimmte Situationen anwenden zu können. Damit soll die Komplexität der Planung möglichst niedrig gehalten werden.

In den folgenden Unterabschnitten wird untersucht, wie Szenarios aufgebaut werden können.

#### 3.3.1. Systematische Darstellung

Im Unterabschnitt 3.2.1 wurden die zu den Userstories gehörenden Ziele aufgelistet. Um aus den Zielen für Userstory 1 ein Szenario zu erstellen, werden im ersten Schritt die Ziele jeweils einem der drei Zieltypen zugeordnet und anschließend zeitlich relativ zueinander angeordnet. Abbildung 3.3 zeigt das Ergebnis dieses Schritts.

In dem Graphen stellt jede horizontale Gerade ein Ziel dar. Die grauen und schwarzen Ziele zusammengenommen bilden die Userstory 1 ab.

Es ist zu erkennen, dass die Ziele sich zum Teil überlappen oder sogar während der Erfüllung eines anderen Ziels erfüllt werden müssen. Als Beispiel: Der Bewohner soll über den Tagesablauf

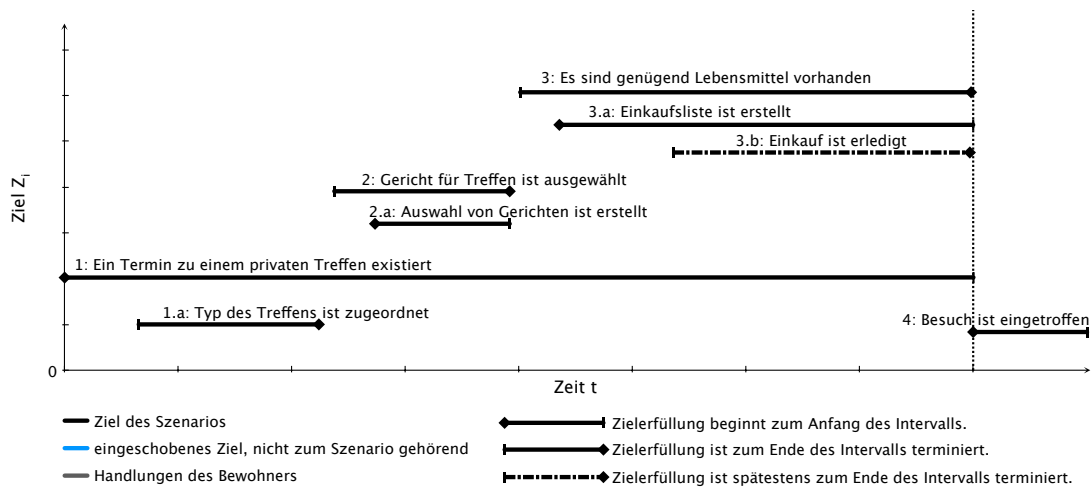


Abbildung 3.4.: Erste Skizze des Szenario-Skripts für Userstory 2 von Seite 22.

informiert werden. Dieses Ziel fängt nicht nur an während der Bewohner frühstückt, sondern es muss erfüllt werden, während auch die Unterlagen bereitgestellt werden.

Wie schon erwähnt, stellt dieser Graph keine feste zeitliche Zuordnung dar, sondern die Ordnung der Ziele relativ zueinander. Das Szenario an sich ist unabhängig von festen Zeitpunkten. Das Szenario beschreibt den Tagesbeginn und konkrete Zeiten ergeben sich erst dann, wenn das Szenario auf eine konkrete Situation angewendet wird. Wie in Abschnitt 3.1.3 beschrieben, sind alternative Szenarios für einen Tagesbeginn denkbar.

Wird dieser Schritt auch für Userstory 2 vollzogen, so ergibt sich eine geradlinige Übertragung der Ziele aus Abschnitt 3.2.1 in einen Graphen (siehe Abb. 3.4). Das Ziel für den Termin stellt das wichtigste Ziel dar. Wird der Termin gelöscht, so ist dieses Szenario hinfällig, daher muss es bis zum konkreten verabredeten Besuchstermin existieren. Die Hilfsziele stehen in diesem Graphen gleichberechtigt neben den Hauptzielen. Wie schon in Abschnitt 3.2.1 erwähnt, müssen diese nur dann geplant werden, wenn die Hauptziele sonst nicht erfüllt werden können.

Die Darstellungen 3.3 und 3.4 geben einen ersten Überblick, wie das Szenario gestaltet sein muss.

In einem zweiten Schritt werden weitere Eigenschaften von Zielen berücksichtigt.

### 3.3.2. Obligatorischer Pfad

Orthogonal zu den Zieltypen können aus Sicht der Szenarios den Zielen weitere Eigenschaften zugeschrieben werden. Es werde obligatorische und optionale Ziele unterschieden.

### 3. Analyse

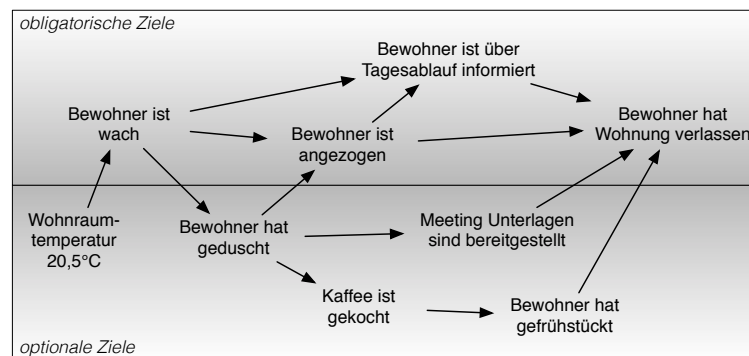


Abbildung 3.5.: obligatorische und optionale Ziele von Userstory 1

Einem konkreten Szenario stehen Situationen entgegen, in denen der Bewohner seine Gewohnheiten fallen lässt. In so einem Fall kann das Szenario hinfällig werden. Dies ist davon abhängig, welche angenommenen Handlungen ausbleiben. In Userstory 1 könnte der Bewohner das Frühstück zuhause ausfallen lassen, dafür unterwegs etwas beim Bäcker kaufen und stattdessen die bereitgestellten Unterlagen fürs Meeting in Ruhe bei einem Kaffee durchsehen. In diesem Fall ändert sich an dem Szenario wenig, außer, dass der Bewohner die Wohnung eventuell etwas früher verlassen muss. Das Frühstück stellt in diesem Zusammenhang ein *optionales* Ziel dar. Optionale Ziele zeichnen sich dadurch aus, dass sie für das Szenario nicht bindend sind. Sie können erfüllt werden, müssen es aber nicht.

In Userstory 2 hingegen führt das Ausschlagen der Auswahl eines Gerichts dazu, dass das Szenario hinfällig wird. Ohne die Auswahl eines Gerichtes durch den Bewohner kann das System keine Zutatenliste und damit keine Einkaufsliste erstellen. Ein Gericht auszuwählen stellt in diesem Szenario ein *obligatorisches* Ziel dar.

Obligatorische Ziele zeichnen sich dadurch aus, dass sie für das Szenario von essentieller Bedeutung sind. Wird eines dieser Ziele nicht erreicht, ist das Szenario hinfällig und ein anderes zu der Situation passendes muss gefunden werden. Die Erfüllung aller obligatorischen Ziele eines Szenarios bilden den obligatorischen Pfad.

Angewendet auf Userstory 1 lässt sich feststellen, dass um das Tagesbeginn-Szenario erfolgreich abschließen zu können, der Bewohner zumindest wach sein muss, zudem sollte er sich anziehen und über den Tagesablauf informiert sein, bevor er die Wohnung verlässt. Das verlassen der Wohnung stellt dabei ebenso ein obligatorisches Ziel da. Denn wenn der Bewohner das nicht tut, wird er zu spät oder überhaupt nicht zu seinem ersten Termin erscheinen und es wäre ein anderes Szenario.

### 3. Analyse

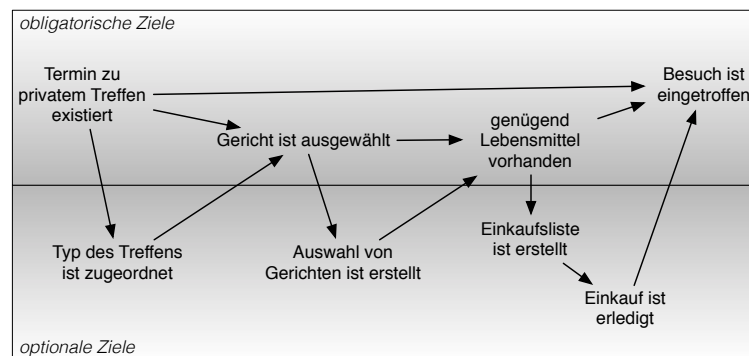


Abbildung 3.6.: obligatorische und optionale Ziele von Userstory 2

Diesen Überlegungen folgend können die Ziele des Szenarios (Abb. 3.3) in den Graphen in Abbildung 3.5 übertragen werden.

Die Pfeile geben die Ordnung zwischen den Zielen wieder. Das Ziel „Wohnraumtemperatur beträgt 20, 5°C“ ist zeitlich vor dem Ziel „Bewohner ist wach“ angesiedelt.

Mehrere ausgehende Pfeile aus einem Knoten bedeuten, dass die Nachfolger aus Sicht dieses Knotens parallel ablaufen könnten. Auf das Ziel „Bewohner ist wach“ folgen „Bewohner hat geduscht“ und „Bewohner ist angezogen“. Da die letzten beiden nicht parallel erfüllt werden können, müssen diese beiden Ziele serialisiert werden, wenn der Bewohner sich vor dem Anziehen wirklich duscht. In diesem Fall sollte man annehmen, dass der Bewohner das schon richtig macht. Handelt es sich bei den Nachfolgern um Ziele, die nicht der Bewohner, sondern Agenten des Systems erfüllen und durch Pfeile miteinander verbunden und damit in eine zeitliche Ordnung gebracht worden sind, müssen sie nacheinander erfüllt werden.

Mehrere eingehende Pfeile in einen Knoten zeigen prinzipiell an, dass das Ziel nur dann erfüllt werden soll, wenn alle vorherigen Ziele erfüllt wurden. Ausnahme bilden z. B. bei dem Ziel „Bewohner hat Wohnung verlassen“ die optionalen Ziele, denn der Bewohner kann ohne Weiteres die Wohnung verlassen ohne gefrühstückt oder die Unterlagen angesehen zu haben, allerdings nicht ohne wach, angezogen und über den Tag informiert worden zu sein. Genau genommen kann er die Wohnung auch verlassen, wenn er nicht informiert wurde, doch in einem solchen Fall hat dieses Szenario keine Gültigkeit mehr. An dessen Stelle muss ein anderes Szenario treten, z. B. eines, das den Arbeitsweg mit den Zielen „Bewohner hat gefrühstückt“, „Bewohner ist über Tagesablauf informiert“ usw. abdeckt.

Analog zu diesen Überlegungen können die Ziele des Szenarios aus Abbildung 3.4 in den Graphen in Abbildung 3.6 übertragen werden.

Wie schon in Abbildung 3.4 verdeutlicht, muss der Besuchstermin bis zum Eintreffen des Besuchs existieren (bzw. erfüllt sein), daher gibt es eine direkte Verbindung von diesem Knoten in den Endknoten. Parallel dazu werden die Ziele „Gericht ist ausgewählt“ und „ausreichend Lebensmittel vorhanden“ erfüllt.

Die Hilfsziele werden hier als optionale Ziele definiert. Somit sind sie in diesem Graphen nicht von anderen optionalen Zielen zu unterscheiden. Wichtig ist hierbei, dass diese Ziele erst dann benötigt werden, wenn das Hauptziel nicht erfüllt werden kann. Es ist also notwendig einen Algorithmus zu entwickeln, in dem Haupt- und Hilfsziele derart verknüpft sind, dass ein Hilfsziel, nur mit einem Hauptziel auftreten kann, dagegen ein Hauptziel auch ohne Hilfsziel.

#### 3.3.3. Einsatz der Intervallalgebra

In Abschnitt 3.2.2 wurde untersucht, wie Ziele mit Hilfe der Intervallalgebra in Beziehung gesetzt werden können. Mit diesem Wissen und aufbauend auf den letzten Betrachtungen können nun die Szenarios als Netze von Intervallen beschrieben werden, dessen Intervalle über Intervallrelationen verknüpft sind.

Werden als Beispiel die beiden Ziele „Wohnraumtemperatur beträgt 20, 5°C“ ( $Z_1$ ) und „Bewohner ist wach“ ( $Z_2$ ) aus der ersten Userstory herangezogen, so lässt sich über ihre Beziehung zueinander sagen:

$Z_1$  muss *vor* oder *überlappend* zu  $Z_2$  erfüllt werden

Denn  $Z_1$  ist in Abbildung 3.3 als Zieltyp 2 definiert, das bedeutet, es lässt sich folgende Relation bilden:

Wohnraumtemperatur beträgt 20, 5°C  $\xrightarrow{\{b,m,o\}}$  Bewohner ist wach

Werden die übrigen Ziele aus Userstory 1 und 2 ebenso umgewandelt, dann erhält man die in den Abbildungen 3.7 und 3.8 dargestellten Szenarios.

Es ist nicht notwendig jeden Knoten mit jedem anderen in Beziehung zu setzen, denn die fehlenden Beziehungen können über den transitiven Zusammenhang ermittelt werden<sup>4</sup>.

### 3.4. Zusammenfassung

Anhand der Userstories 1 und 2 werden Ziele und Aktionen ermittelt. Ziele sind gewünschte Zustände einer Ressource zu einem bestimmten Zeitpunkt oder für eine konkrete Zeitspanne. Mit

---

<sup>4</sup>siehe dazu Allen (1983) S. 834ff und Nau u. a. (2004) S. 293ff

### 3. Analyse

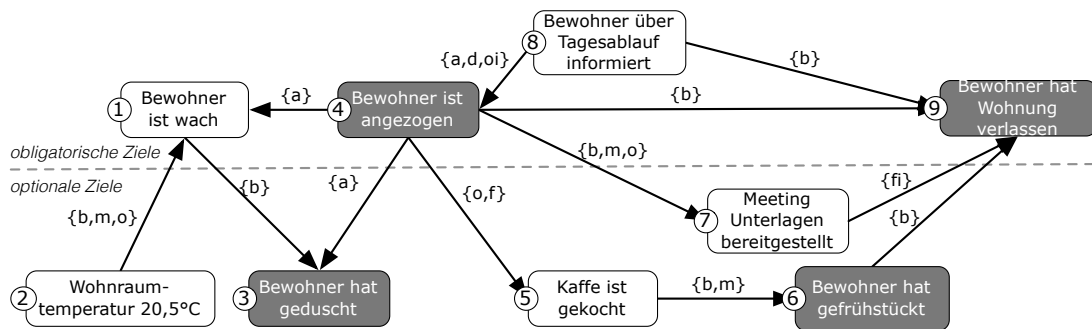


Abbildung 3.7.: Ziele von Userstory 1 mittels der Intervalalgebra in Beziehung gesetzt.

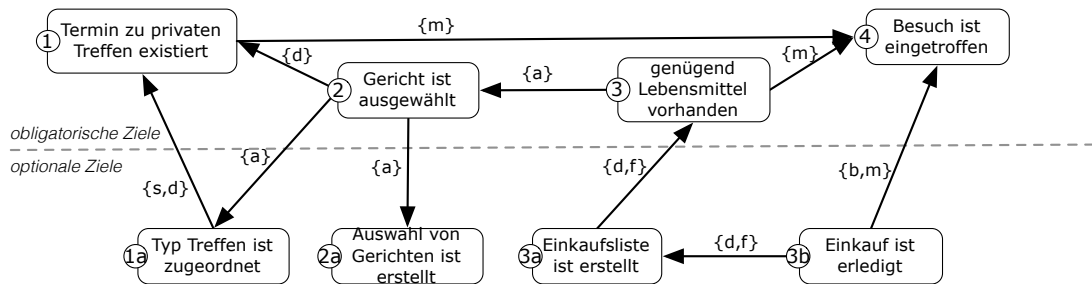


Abbildung 3.8.: Ziele von Userstors 2 mittels der Intervalalgebra in Beziehung gesetzt.

Hilfe von Aktionen können diese Zielzustände erreicht werden. Anders als klassisch planende Systeme, wird das hier dargestellte System keine Aktionen, sondern Ziele planen. Die Aktionen entstehen implizit, wenn das Gesamtsystem versucht die Ziele zu erreichen.

Ziele sind Entitäten mit einer zeitlichen Ausdehnung und können als Zeitintervalle dargestellt werden. Die Ziele können in drei verschiedene Typen unterteilt werden.

1. Zielerfüllung beginnt am Anfang des Intervalls und besteht für das ganze Intervall.
2. Zielerfüllung ist zum Ende des Intervalls terminiert.
3. Zielerfüllung ist spätestens zum Ende des Intervalls terminiert.

Ziele werden als folgendes Tupel dargestellt:

$$Z = (Was, Wo, Wer, Wann, Wie lang, RelationZu)$$

Den Userstories werden Ziele zugeordnet und diese mit Hilfe von Intervallrelationen zu Szenarios zusammen gefügt, so dass sie die Userstories als Szenarios abbilden.

Zudem wird festgestellt, dass es optionale und obligatorische Ziele gibt. Optionale Ziele können unerfüllt bleiben, ohne dass das Szenario abgebrochen wird. Obligatorische Ziele hingegen müssen erfüllt werden, sonst gilt das Szenario als verlassen. Der obligatorische Pfad muss eingehalten werden.

### 3.5. Erkenntnisse der Analyse

Im diesem Kapitel wurde eine Definition von Zielen erarbeitet und anschließend Ziele zu Szenarios zusammengefügt, um sozusagen Lebenssituationen abzubilden. Daraus ergeben sich die Fragen nach dem Nutzen dieses Wissens und in wie fern es dem System einer intelligenten Wohnung hilft, den Bewohner in seinem Alltag zu unterstützen.

Wenn die Lebensgewohnheiten des Bewohners der intelligenten Wohnung bekannt sind, muss es möglich sein, zu Beginn so einer gewohnten Situation zu erkennen, dass der Bewohner sich in diese begibt.

Gelingt bei Eintritt das Erkennen dieser Lebenssituation, bzw. deren Zuordnung zu einem Szenario, hinreichend schnell, dann können die Szenarios als Vorhersage dienen, was als nächstes passieren und was wichtig sein wird. Mit anderen Worten steht damit der kontextuelle Moment nicht für sich allein, sondern im Zusammenhang eines größeren Kontextes, dessen Bedeutung und Struktur in Grundzügen als Szenario bekannt ist.



Dies ist der Übergang von Behaviour-Interpretation zur Behaviour-Prediction.

Das ermöglicht Ressourcen rechtzeitig zu binden und zu reservieren, und verhindert, dass Aufgaben mit geringerer Priorität diese Ressourcen eventuell kurz vorher beanspruchen und nicht mehr rechtzeitig freigeben.

Mit anderen Worten: Die Ziele haben synchronisierende und koordinierende Funktion in Bezug auf kleine Strukturen. Szenarios haben synchronisierende und koordinierende Funktion bezogen auf größere Strukturen.

Ohne diese Strukturen müssten für mehrere Agenten separat Regeln definiert werden, nach denen sie jeweils agieren. Agenten können in diesem Zusammenhang bestimmte Geräte sein. Kommt ein neues Gerät dazu oder entfällt eines, müssen die Regeln wieder angepasst werden. Das Abstimmen eines solchen Systems auf mehrere Situationen kann sich als sehr komplex erweisen und unpraktikabel sein.

Szenarios bilden Lebenssituationen relativ simpel ab. Sie sind zentrale Schablonen, die zeitliche Abhängigkeiten und Bedingungen der jeweiligen Lebenssituationen beinhalten. Diese Schablonen gelten für alle beteiligten Agenten und existieren unabhängig von den gerade verfügbaren Agenten im System. Sie müssen nicht angepasst werden, wenn neue Agenten dem System hinzugefügt oder aus dem System entfernt werden.

### 3.6. Ermittelte Anforderungen an das Design

Nach den Erkenntnissen der Analyse lassen sich die für das Zielsystem benötigten Komponenten bestimmen. Aus Low-Level-Sensordaten werden semantische Informationen gewonnen. Diese Daten müssen in mehreren Schritten so weit verdichtet werden, dass Aussagen über Szenarios getätigt werden können.

Damit ergibt sich folgender Ablauf:

1. *Kontextwissen erstellen*: Die Sensordaten müssen semantisch interpretiert werden.
2. *Szenario auswählen*: Aus dem Kontext muss auf ein Szenario geschlossen werden.
3. *Ziele verteilen*: Die aus dem Szenario resultierenden Ziele müssen den Komponenten des Systems bekannt gemacht werden.
4. *Ziele erfüllen*: Die Erfüllung der Ziele muss im Bereich der Möglichkeiten der einzelnen Komponenten liegen.

Abbildung 3.9 zeigt ein abstraktes Vorgehensmodell, das diesen Ablauf visualisiert. Konkret müssen die Sensorwerte aus dem Living Place einen mehrstufigen Interpretationsprozess durch-

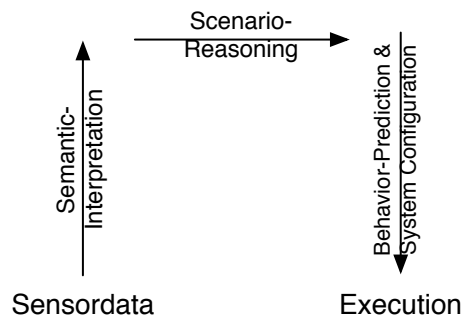


Abbildung 3.9.: abstraktes Vorgehensmodell

laufen, bei dem die Werte mit Semantik angereichert werden. Am Ende dieses Prozesses wird ein passendes Szenario ermittelt. Ist ein Szenario erkannt worden, so kann eine Vorhersage für die zeitlich naheliegenden Tätigkeiten des Bewohners getroffen und die Gerätelandschaft des Living Places dementsprechend konfiguriert werden.

Die grundlegende Systemarchitektur des Living Places ist in [Ellenberg u. a. \(2011\)](#) beschrieben. Die Bereitstellung und Gewinnung höherer Kontextinformation wird z.B. in der Arbeit von [Ellenberg \(2011b\)](#) untersucht. Die vorliegende Arbeit konzentriert sich auf die Darstellung, Verteilung und Erfüllung der Ziele im Smart-Home. Auf die Abbildung 3.9 bezogen, bedeutet dies, es geht um die Bereitstellung der Systemziele, die Verteilung dieser Ziele an Agenten, die diese Ziele erfüllen können, und die Koordination dieser Agenten.

Zuerst muss aus den semantischen Kontextinformationen ein Szenario hergeleitet werden. Anschließend müssen Ziele aus diesem Szenario ausgewählt werden, die erfüllt werden sollen. Bereits erfüllte Ziele sollen nicht erneut bekannt gemacht werden. Ziele, die erst zu einem späteren Zeitpunkt relevant sind, sollen noch nicht bekannt gemacht werden. Zudem muss sichergestellt werden, dass für bekannt gemachte Ziele jeweils mindestens ein Agent existiert, der das Wissen besitzt es zu erfüllen.

Aufgrund dieser Überlegungen können die oben genannten Anforderungen für diese Arbeit folgendermaßen konkretisiert werden:

1. *Darstellung von Szenarios*: Die Szenarios müssen in ihrer Gesamtheit dargestellt werden können, das heißt, die sie umfassende Ziele und die Relationen zwischen diesen Zielen. Außerdem soll ein Pfad von obligatorischen Zielen definiert werden können.
2. *Verteilung der Ziele*: Die Ziele müssen jeweils bei den Agenten ankommen, die für diese Ziele verantwortlich sind. Dies kann zum Beispiel erfolgen indem die Agenten einen Nachrichtentyp abonnieren (in einer messageorientierten Architektur).

### 3. Analyse

---

3. *Koordination der Erfüllung*: Für die Erfüllung von Zielen müssen die Agenten sich koordinieren können. Dies ist zum Beispiel notwendig, falls mehr als ein Agent ein Ziel übernehmen kann.
4. *Zeitliche Ausrichtung der Ziele*: Die genauen Zeitpunkte zur Erfüllung der Ziele ergeben sich zum Teil erst während der Laufzeit. Es muss ein Mechanismus existieren, der es erlaubt, dass die Agenten zur Laufzeit zeitliche Grenzen propagieren und untereinander verhandeln können.

Anhand dieser Anforderungen wird in Kapitel 4 ein Konzept für eine Systemkomponente entwickelt, die diese .

## 4. Design und Realisierung

In diesem Kapitel wird auf Grundlage der Analyse und der aufgestellten Anforderungen aus dem vorherigen Kapitel ein Design für eine Systemkomponente erarbeitet, die aus einem erkannten Szenario aktuell zu erfüllende Ziele ermittelt und diese den Agenten des Smart-Homes bekannt macht, damit diese die Ziele erfüllen. In den ersten Abschnitten (Abschnitte 4.1 - 4.5) wird sich dem Konzept des Designs gewidmet, und danach (Abschnitte 4.6 - 4.9) die Implementierung eines Prototyps umrissen, welcher in Scala implementiert wird. Anschließend (Abschnitt 4.10) werden Tests mit diesem Prototypen durchgeführt und die Ergebnisse ausgewertet. Abschließend (Abschnitt 4.12) wird das Gesamtkonzept der Systemkomponente evaluiert.

### 4.1. Systembeschreibung

Nach der Festlegung der Anforderungen für die in dieser Arbeit zu designende Systemkomponente in Abschnitt 3.6, werden in diesem Abschnitt die Funktionseinheiten identifiziert, die vor dieser Systemkomponente liegen, und deren Aufgaben in Hinblick auf die für die Systemkomponente notwendigen Informationen erläutert. Als letztes wird die hier behandelte Systemkomponente vorgestellt.

Wie die Abbildung 4.1<sup>1</sup> verdeutlicht, werden aus den eingehenden rohen Sensorinformationen durch Interpretation semantische Aussagen geschlossen („Person A ist wach“, „Es ist vormittag“ etc.). Anschließend werden in einem weiteren Schritt die semantischen Aussagen auf eine höhere Abstraktionsebene gehoben und auf Szenarios abgebildet. Ist ein Szenario erkannt worden, werden ihm eine Menge von Zielen zugeschrieben. Diese Ziele werden an Agenten übermittelt, die diese dann erfüllen sollen. In den folgenden Unterkapiteln werden die einzelnen Funktionseinheiten genauer beleuchtet.

#### 4.1.1. Sensordaten-Interpretation

Voskuhl (2010) beschreibt die Breitstellung einer Sensorwolke, die die Werte aller Sensoren, egal ob low-level oder high-level, bereitstellt. Dieser Plan wird in Ellenberg u. a. (2011) koncreti-

---

<sup>1</sup>Angelehnt an Abb. 1 aus Neumann und Terzic (2011).

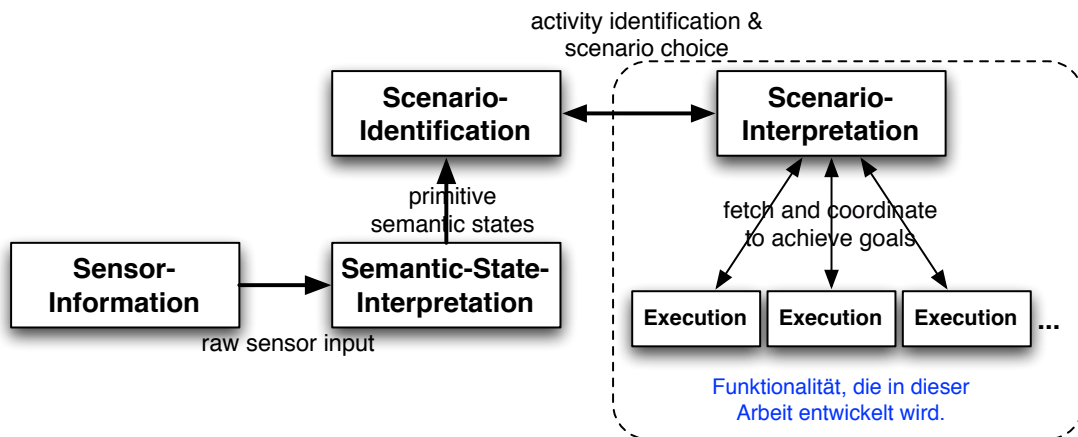


Abbildung 4.1.: Schematische Darstellung der Funktionseinheiten zur Interpretation von Sensorinformationen und der Identifikation und Interpretation von Szenarios, sowie das Erfüllen von Zielen.

siert und in ein Architekturmodell überführt. Diese Funktionalität wird in Abbildung 4.1 als Sensor-Information bezeichnet. Desweiteren wird in Karstaedt (2011b) und Karstaedt (2011a) das Verdichten dieser Informationen mittels eines 3D Modell des Living Places beschrieben. Das 3D-Modell dient zur Ermittlung von räumlichen Informationen z. B. über *Funktional Spaces* von Geräten, also Bereichen, von denen aus man ein Gerät bedienen kann, oder *Operational Spaces*, Bereiche in denen ein Gerät agiert, z. B. der Schwingbereich einer Tür<sup>2</sup>. Dieser Prozess der mehrstufigen Anreicherung von Low-Level-Kontextinformationen mit höheren semantischen Aussagen wird in Abbildung 4.1 als Semantic-State-Interpretation bezeichnet. Die Ausarbeitung der Semantic-State-Interpretation ist derzeit noch nicht abgeschlossen, Ergebnisse aus dazu laufenden Projekten und Masterarbeiten werden erwartet.

#### 4.1.2. Scenario-Reasoning

Die zu erkennenden Szenarios müssen dem System bereits vorher bekannt sein. Aufbauend auf den semantischen Sensordaten-Interpretationen werden High-Level-Kontextinformationen generiert, die auf Aktivitäten abgebildet werden. Zum Beispiel, dass eine Person kocht, fernsieht oder sich anzieht. Diese Aktivitäten werden dann auf Szenarios zurückgeführt. Diese Funktionalität wird in Abbildung 4.1 als Scenario-Identification bezeichnet. Das Softwaremodul, das diese Funktionalität bietet, wird im Folgenden Scenario-Reasoner genannt. Ellenberg (2011b) behandelt

<sup>2</sup>Diese Informationen werden in der Ontologie, die in der Masterarbeit von Ellenberg (2011b) erarbeitetet wird, abgebildet.

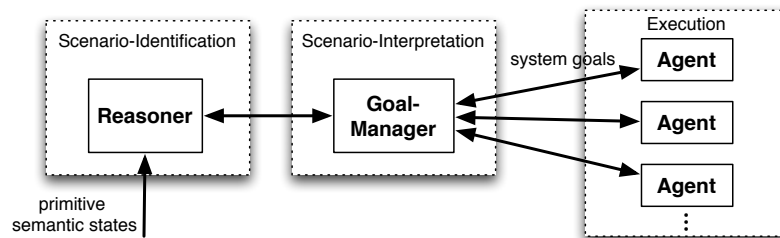


Abbildung 4.2.: Die Funktionalitäten der Scenario-Identification und -Interpretation werden von den beiden Softwaremodulen Reasoner und Goal-Manager übernommen.

im Rahmen seiner Masterarbeit das Scenario-Reasoning und baut dabei auf die Erkenntnisse des CoFriend-Projekts von [Bohlken u. a. \(2011\)](#) auf.

#### 4.1.3. Behaviour-Prediction und System-Configuration

Nachdem ein Szenario erkannt wurde, wird dieses interpretiert, indem dem Szenario eine Menge von Zielen zugeordnet wird, und diese Ziele den Agenten des Systems nach und nach zur Erfüllung zur Verfügung gestellt werden. Dieser Vorgang wird in [Abbildung 4.1](#) als Scenario-Interpretation bezeichnet. Diese Funktionalität wird vom Softwaremodul Goal-Manager übernommen, dessen Aufbau in dieser Arbeit im [Abschnitt 4.3](#) entwickelt wird.

[Abbildung 4.2](#) zeigt den Scenario-Reasoner und Scenario-Interpreter in einem Modul. Dieses Modul beinhaltet die beiden Komponenten Reasoner und Interpreter. Logisch bilden beide Komponenten eine Einheit und arbeiten zusammen. Hat der Reasoner der Lebenssituation ein Szenario zugeordnet, teilt er dies dem Interpreter mit, der daraufhin zu der Lebenssituation passende Ziele ermittelt. Je nachdem wie weit das Szenario fortgeschritten ist, teilt er den Agenten nur die zur Zeit relevanten Ziele mit. Damit wird das Verhalten des System für das konkrete Szenario konfiguriert (System-Configuration). Das Auswählen von Zielen stellt eine Verhaltensvorhersage (Behaviour-Prediction) dar.

### 4.2. Architektur

[Abbildung 4.3](#) verdeutlicht welche Komponenten des Living Places am in [Abschnitt 4.1](#) beschriebenen Gesamtprozess beteiligt sind. Die hell dargestellten Komponenten sind gegeben und wurden bzw. werden in anderen Arbeiten und Projekten untersucht. Sie wurden im Grundlagen-

kapitel (2.3) vorgestellt. Die dunklen Komponenten werden im Rahmen der vorliegenden Arbeit entwickelt.

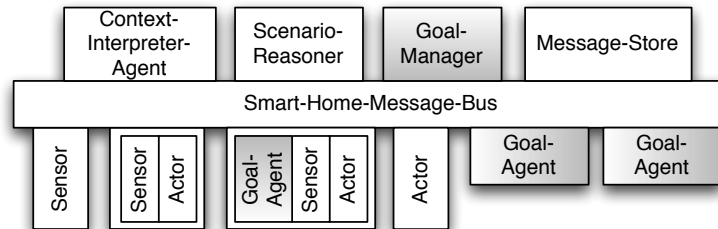


Abbildung 4.3.: Architektur des Living Places inklusive des Goal-Managers und Goal-Agents

Analog zu dem Vorgehensmodell aus Abbildung 3.9 wird in diesem Architekturmodell die Sensordaten-Interpretation vom Context-Interpreter-Agent übernommen. Das Scenario-Reasoning wird vom Scenario-Reasoner durchgeführt. Das Scenario-Interpretation übernimmt der Goal-Manager, dessen Struktur in Abschnitt 4.3 entwickelt wird.

Sensoren und Aktoren sind die Geräte im Living Place oder andere Softwarekomponenten (z.B. ein Wetterinformationsagent). Sensoren liefern aus Sicht der vorliegenden Arbeit Low-Level-Kontextinformationen. Aktoren können Steuerbefehle empfangen und diese in Aktionen umsetzen. Die Goal-Agents, auf welche in Abschnitt 4.4 eingegangen wird, können solche Steuerbefehle senden, um Ziele zu erfüllen. Goal-Agents, Sensoren und Aktoren müssen nicht getrennt voneinander existieren, sondern können ein und dasselbe Gerät bzw. ein und dieselbe Softwarekomponente darstellen.

### 4.3. Goal-Manager

Der Goal-Manager besitzt Repräsentationen der möglichen Szenarios und erhält vom Scenario-Reasoner die Information, in welchem Szenario sich das Smart-Home befindet, und welche der Ziele des aktuell ausgewählten Szenarios inzwischen erfüllt sind. Aus diesen Informationen soll der Goal-Manager schließen, welche Ziele als nächstes erfüllt werden sollen. Wenn die nächsten Ziele ermittelt sind, macht der Goal-Manager diese dem Smart-Home bekannt und plant sie ein. Anschließend können Goal-Agents die Erfüllung dieser Ziele übernehmen. Es geht bei diesem Design des Goal-Managers nicht darum einen vollwertigen Planer zu entwickeln, sondern eine Straight-Forward-Methode zu erarbeiten, wie Folgeziele gefunden werden können, um die generelle Machbarkeit des Designkonzepts zu untersuchen.

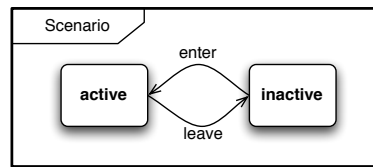


Abbildung 4.4.: Zustände eines Szenarios

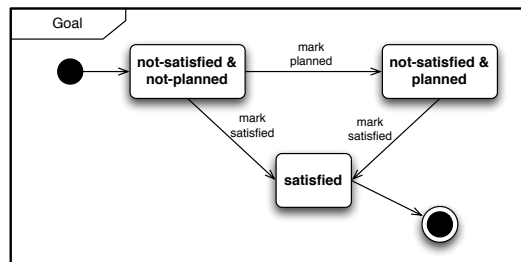


Abbildung 4.5.: Zustände eines Zieles

In Abschnitt 3.3 der Analyse werden Szenarios als Graphen, deren Kanten mit Allen-Intervall-Relationen gewichtet sind, dargestellt. Da der Goal-Manager weiß, welche Knoten dieses Graphen positiv markiert sind, also welche Ziele erfüllt sind, kann er über eine Graphsuche die Folgeknoten finden. Im Folgenden wird nun erläutert, wie zu planende Ziele mittels Graphensuche gefunden werden sollen.

#### 4.3.1. Zustände von Szenarios und Zielen

Grundlegend sind Szenarios und Ziele in der Analyse in Abschnitt 3.2 und 3.3 beschrieben worden. An dieser Stelle sollen den Szenarios Eigenschaften zugeordnet werden, die für das Design des Goal-Managers notwendig sind.

Ein Szenario besitzt Vorbedingungen, damit es aktiv werden kann, das heißt, es kann aktiv oder inaktiv sein. Dies zeigt das Zustandsdiagramm 4.4. Ein Szenario ist eine Struktur von zeitlich relativ zueinander geordneten Zielen. Ein aktives Szenario enthält eine Menge von Zielen, die mit den Handlungen und Aktionen einer bestimmten Lebens- oder Wohnsituation korrespondiert. Die Ziele existieren in einer relativen zeitlichen Struktur zueinander. Sie stellen Intentionen dar, die mit bestimmten Bewohnern oder Geräten der Wohnung verbunden sind.

Ziele besitzen Informationen darüber, welche Ressourcen sie betreffen, welchen Zustand diese Ressourcen einnehmen sollen und welchen Zieltyp sie repräsentieren. Ein Ziel kann erfüllt



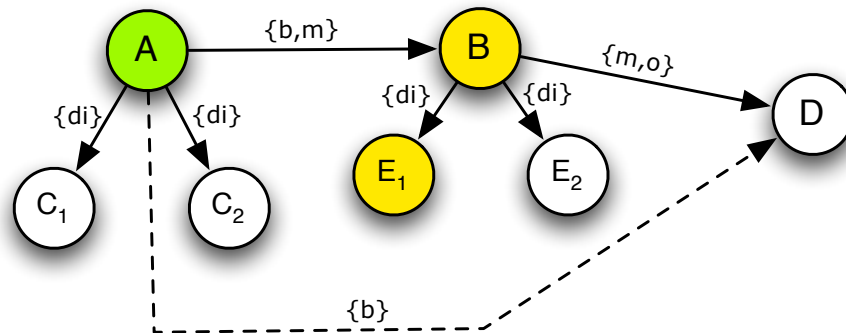


Abbildung 4.6.: Szenario-Graph des Goal-Managers.

und nicht erfüllt sein und orthogonal dazu geplant und nicht geplant sein. Ein nicht erfülltes Ziel kann geplant werden. Ein erfülltes Ziel darf nicht geplant werden. Diese Regel wird im Zustandsdiagramm 4.5 abgebildet.

#### 4.3.2. Ziele finden mittels Graphensuche

Abbildung 4.6 zeigt einen Beispielgraphen eines Szenarios. Grün markierte Knoten sind erfüllte Ziele, gelb markierte Knoten stellen geplante Folgeziele dar. In geschweiften Klammern stehen Zeitintervall-Relationen, welche die Kanten gewichten. Ziele zu denen es Alternativen gibt, werden mit dem selben Buchstaben wie die Alternativziele gekennzeichnet, allerdings jeweils mit unterschiedlichem Index.

Knoten  $A$  ist ein erfülltes Ziel. Folgeknoten zu  $A$  sind  $B$  und  $E$ . Knoten  $D$  liegt außerhalb der unmittelbaren Erreichbarkeit von  $A$ , da die transitive Relation  $AD$  die Gewichtung  $\{b\}$  besitzt. Dies wird weiter unten als außer Reichweite definiert. Denn entscheidend sind nicht die Kanten im Graphen, sondern die über die Intervallalgebra definierten oder transitiven zeitlichen Relationen zwischen allen Knoten. Die  $C$ -Knoten werden nicht gelb markiert, da durch die Relation  $di$  die Erfüllung eines der Alternativziele  $C_i$  während der Erfüllung des Ziels  $A$  hätte passieren müssen, aber nicht geschehen ist. Die Bewertung, ob das Szenario durch das Wegfallen von  $C$  noch besteht, muss der Szenario-Reasoner durchführen. In diesem Beispiel besteht das Szenario weiterhin. Anders bei den Alternativzielen  $E_i$ . Über die Relation  $di$  sollen diese erfüllt werden, während  $B$  erfüllt wird.  $B$  folgt auf  $A$ , durch die transitive Relation folgen auch die Ziele  $E_i$  auf  $A$ . Da  $B$  nicht erfüllt ist und eingeplant werden soll, muss jetzt ebenso eines der Ziele  $E_i$  eingeplant werden.

Während der Laufzeit eines Szenarios wird es sehr wahrscheinlich mehrere Knoten geben, die gleichzeitig erfüllt sind. Daher wird der Goal-Manager von allen erfüllten Knoten aus die Suche nach Folgeknoten durchführen. Die Auswahl von zeitlich naheliegenden Zielen als Folgeziele erfolgt, um flexibel auf Variationsmöglichkeiten des Szenarios oder auf Szenario-Wechsel reagieren zu können. Ansonsten würden jedes Mal alle Agenten mit der Erfüllung sämtlicher vorhandener noch nicht erfüllten Ziele eines Szenarios beschäftigt werden und dieses möglicherweise bei einer Veränderung der Gegebenheiten ganz umsonst.

Die Suche nach Folgezielen wird über eine Breitensuche realisiert. Da weniger entscheidend ist einen einzelnen Pfad zu verfolgen, sondern die als nächstes anstehenden Ziele zu finden, welche sich idealerweise im Graphen nahe dem Startknoten befinden sollen.<sup>3</sup>

### 4.3.3. Zeitmodell

Wie im Grundlagenkapitel 2.5.2 erwähnt, ist die vollständige Intervallalgebra (Klasse  $\mathcal{IA}$ ) von Allen nicht effizient entscheidbar. Daher wird hier nicht die vollständige Klasse  $\mathcal{IA}$  verwendet. Stattdessen wird eine Unterklasse von  $\mathcal{IA}$  ausgewählt, die ausreicht, um die Ziele eines Szenarios partiell zeitlich zu ordnen.

Im Grundlagenkapitel 2.5.2 wurde die Klasse  $\mathcal{IA}_c$  aus Nau u. a. (2004) beschrieben, die über die Konvexitätsbedingung hergeleitet wird. Es besteht das Ziel, ein Szenario ausreichend zu beschreiben. Dazu ist es nicht notwendig ein vollständiges Constraint-Net aufzubauen. Das Szenario muss nur soweit beschrieben werden, dass vorher-/nachher-Beziehungen zwischen Zielen, die zeitlich nahe beieinander liegen, ermittelt werden können. Alle weiteren Beziehungen können über die transitive Hülle geschlossen werden. Der Aufbau des Graphen muss in Bezug auf die transitive Hülle aller vorhandenen Relationen konsistent sein. Die Klasse  $\mathcal{IA}_c$  besteht aus 82 verschiedenen Relationen. Für die Beschreibung eines Szenarios sind das mehr als gebraucht werden. Mit Beachtung der Konvexitätsbedingung, wird daher hier eine Unterklasse  $G$  von  $\mathcal{IA}_c$ , als das hier zu verwendende Zeitmodell, definiert:

$$G = \left\{ r \mid r \cap \{b, m, o, di, fi\} \neq \emptyset \Rightarrow r \subseteq \{b, m, o, di, fi\} \right\} \quad (4.1)$$

Gleichung 4.1 zeigt die Definition der Menge  $G$ . Diese besteht aus den Relationen  $b, m, o, di, fi$ . Die transitive Eigenschaft von  $G$  ist in Tabelle 4.1 dargestellt. Es zeigt sich, dass die Transitivität von jedem beliebigen Paar eine oder mehrere Relationen der Menge  $G$  ergibt.

---

<sup>3</sup>Um das Konzept zu überprüfen, könnte es in ein Petrinetz überführt werden und z. B. auf Schaltbarkeit und Deadlock / Livelock Eigenschaften untersucht werden.

<i>B r C</i>	<i>b</i>	<i>di</i>	<i>o</i>	<i>m</i>	<i>fi</i>
<i>A r B</i>					
<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>
<i>di</i>	<i>b, o, m, di, fi</i>	<i>di</i>	<i>o, di, fi</i>	<i>o, di, fi</i>	<i>di</i>
<i>o</i>	<i>b</i>	<i>b, o, m, di, fi</i>	<i>b, o, m</i>	<i>b</i>	<i>b, o, m</i>
<i>m</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>
<i>fi</i>	<i>b</i>	<i>di</i>	<i>o</i>	<i>m</i>	<i>fi</i>

Tabelle 4.1.: Transitivitätstabelle für die Intervallklasse  $G$ .

#### 4.3.4. Regeln für die Auswahl von Folgezielen

Im ersten Schritt werden alle Ziele eingeplant, hier gelb markiert, die zu einem erfüllten Ziel mit einer der folgenden Relationen direkt oder transitiv in Relation stehen:  $o, m$ . Denn diese Ziele folgen zeitlich direkt auf das erfüllte Ziel. Mit der Relation  $o$  überschneiden sich diese sogar, das wird als Grenzfall aber akzeptiert.

Im zweiten Schritt werden alle Ziele eingeplant, gelb markiert, die zu den vorher ermittelten Folgezielen mit eine der folgenden Relationen direkt oder transitiv in Relation stehen:  $o, m, di, fi$ . Denn es sollen alle Ziele, die zum einen innerhalb des Intervalls der Ziele aus dem ersten Schritt erfüllt werden sollen  $di, fi$ , und zum anderen die Ziele, die diesen Zielen überlappend oder direkt im Anschluss folgen  $o, m$ , als Folgeziele gelten.

Die Tabellen 4.2 und 4.3 zeigen diese beiden Regeln in der Übersicht.

<i>A r B</i>	<i>B</i> wird markiert
<i>b</i>	-
<i>di</i>	-
<i>o</i>	x
<i>m</i>	x
<i>fi</i>	-

Tabelle 4.2.: Planen von Zielen mit Abstand  $\geq 1$  zu Knoten A. Markierung bezogen auf transitive Relation von A zu C.

Für das Erstellen des Constraint-Nets wird die Klasse  $G'$  verwendet, die wie folgt definiert ist:

$$G' = \left\{ r \mid \begin{array}{l} r \cap \{b, m, o, di, fi\} \neq \emptyset \Rightarrow r \subseteq \{b, m, o, di, fi\} \\ r \cap \{b\} \neq \emptyset \Rightarrow \{m\} \subseteq r \end{array} \right\} \quad (4.2)$$

Wenn ein Ziel im Constraint-Net die Relation  $b$  zu einem zeitlichen Folgeziel besitzt, dann ist die Kante ebenso mit der Relation  $m$  gewichtet. Das ermöglicht ohne eine Sonderregel

$B r C$	$C$ wird markiert
$b$	-
$di$	x
$o$	x
$m$	x
$fi$	x

Tabelle 4.3.: Planen von Zielen mit Abstand  $\geq 1$  von Knoten B. Markierung bezogen auf transitive Relation von B zu C.

auszukommen, die besagt, dass im Constraint-Net existierende Kanten mit dem Gewicht  $\{b\}$  als Folgeziele gelten.

Dieses Verfahren arbeitet mit eindeutigen Zielen. Um nichtdeterministische Alternativziele berücksichtigen zu können, muss dieses Verfahren sinnvoll erweitert werden, z.B. durch die Einführung von Eintrittswahrscheinlichkeiten, um die wahrscheinlichsten Pfade verfolgen zu können.

#### 4.4. Goal-Agents

Goal-Agents können sowohl reaktiv als auch proaktiv agieren, je nachdem wie komplex die Aufgabe ist, für die sie entworfen werden. Jeder Goal-Agent kann dafür entwickelt werden, für ein oder mehrere Ziele zuständig zu sein.

Ein einfacher Goal-Agent, der z.B. für die Wohnraumtemperatur zuständig ist, wird aktiv, sobald er ein Ziel empfängt, das die Wohnraumtemperatur betrifft. Das Ziel kann z.B. die Umstellung von Nacht- auf Tagestemperatur betreffen. Kennt er die genauen Zeiten noch nicht, wird er auf diese warten oder eine Anfrage diesbezüglich stellen. Sobald die Zeiten bekannt sind, kann er einen Scheduler beauftragen, ihn zur gegebenen Zeit an die Änderung der Temperatur zu erinnern.

#### 4.5. Koordination der Agenten

In diesem Abschnitt wird gezeigt, wie sich Goal-Agents nach dem Auswählen eines Szenarios koordinieren, um die Ziele dieses Szenarios zu erfüllen.

Nach [Bedrouni u. a. \(2009\)](#)<sup>4</sup> gibt es zwei Ebenen der Koordination in Verteilten Systemen zur Problemlösung, zum einen die objektive und zum anderen die subjektive. Die objektive

---

<sup>4</sup>siehe S. 6

Koordination bezieht sich auf die grundlegenden Systemeigenschaften, wie z. B. welches Kommunikationsmedium verwendet wird und wie die Agenten erschaffen und wieder gelöscht werden. Diese werden auch als *Inter-Agent-Aspekte* bezeichnet. Die subjektive Koordination bezieht sich auf die Abhängigkeiten der Agenten untereinander, also wie sie sich koordinieren, um Aufgaben zu lösen. Diese werden als *Intra-Agent-Aspekte* bezeichnet.

In diesem Abschnitt geht es um den zweiten Aspekt, der *Intra-Agent-Koordination*.

Dafür werden zwei grundlegende Fälle unterschieden:

1. Einfacher Fall: Genau ein Agent für ein Ziel. zuständig ist.
2. Komplexer Fall: Mehrere Agenten ein Ziel übernehmen können.

Orthogonal dazu können folgende Fälle unterschieden werden:

1. Ein Agent übernimmt ein Ziel und kann dieses allein erfüllen.
2. Ein Agent übernimmt ein Ziel, dass er nicht allein erfüllen kann.

##### 4.5.1. Agenten übernehmen und erfüllen Ziele

###### Ein Agent übernimmt ein Ziel

Dass zu jedem geplanten Ziel genau ein Agent existiert, stellt den Trivialfall dar. Die Agenten warten darauf, dass ein Ziel geplant wird, dass sie übernehmen können. Sobald ein Ziel geplant wird, kann der jeweilige Agent die Erfüllung des Ziels anstreben. Die einzige Koordinationsleistung, die der Agent dabei erbringen muss, ist in diesem Fall, wenn die Zeitgrenzen seines Ziels noch nicht feststehen.

###### Mehrere Agenten übernehmen ein Ziel

Können mehrere Agenten ein Ziel übernehmen, so müssen sie sich so koordinieren, dass nur einer das Ziel bearbeitet. Im einfachsten Fall wird die Entscheidung per zeitlicher Reihenfolge gefällt. Der erste Agent, der das Ziel übernehmen möchte, bekommt es. Andernfalls müssen Entscheidungskriterien definiert, zentral überprüft oder zwischen den Agenten ausgehandelt werden. Für Multiagent-Planning und die Koordination innerhalb von Multiagenten-Systemen sei auf [de Weerd u. a. \(2005\)](#) und [Bedrouni u. a. \(2009\)](#) verwiesen.

###### Ein Agent kann ein Ziel allein erfüllen

Kann ein Agent ein Ziel allein erfüllen entspricht das dem Trivialfall. Der Agent besitzt zum Erfüllen des Ziels die nötigen Ressourcen oder kann auf diese zugreifen und besitzt das nötige

#### 4. Design und Realisierung



Abbildung 4.7.: Propagieren der Start- und Endzeit zwischen zwei voneinander abhängigen Zielen.

Wissen um es zu erfüllen. Zu den Ressourcen gehören Algorithmen und der Zugriff auf Geräte und Sensoren über z. B. Webservices. Zum Wissen zählt gehört das Kontextwissen des Smart-Homes und Domänenwissen.

#### Ein Agent kann ein Ziel nicht allein erfüllen

Besitzt ein Agent nicht die notwendigen Ressourcen oder das notwendige Wissen, um ein Ziel allein zu erfüllen, kann er das Ziel in Subziele zerlegen und diese seinerseits an andere Agenten verteilen. Das Zeitintervall des Hauptziels ergibt sich aus der Summe der Dauer der Subziele.

#### 4.5.2. Propagieren der Start-/Endzeiten von Zielen

Abbildung 4.7 zeigt zwei Ziele, die in Relation zueinander stehen. Bei  $Ziel_i$  (sowie bei  $Ziel_{i-n}$ ) ist die konkrete Endzeit bekannt ( $t_1$ ). In der Abbildung ist die Dauer für das Lösen des Ziels schon bekannt. Diese ist vorher von dem Goal-Agent ermittelt worden. Im einem nächsten Schritt schlussfolgert der Goal-Agent die Startzeit für  $Ziel_i$  ( $t_2$ ). Da ihm nun Start- und Endzeit für das  $Ziel_i$  bekannt sind, verteilt er dieses Wissen an alle Agenten, die von diesem Ziel abhängige Ziele erfüllen müssen.

Der Agent der  $Ziel_{i-n}$  übernimmt kann anschließend, abhängig von der Relation seines Ziel zum Folgeziel eine Endzeit festlegen ( $t_3$ ). Anschließend berechnet dieser die Startzeit für  $Ziel_{i-n}$  ( $t_4$ ).

Zum Beispiel orientiert sich der Agent, der die Heizung steuert, mit  $Ziel_{i-n}$  an den Zeiten aus  $Ziel_i$ , dessen Zeiten aus dem Terminkalender und gewissen Erfahrungswerten, seinem Usermodell, vom Weckagenten ermittelt wurden.

### 4.6. Technologie

Scala<sup>5</sup>, die auf der Java Virtual Machine basiert, ist eine funktionale-objektorientierte Sprache, deren Ziel es ist die Vorteile beider Welten zu vereinen. Entwickelt wird Scala seit 2001 von einer Gruppe um Martin Odersky an der École Polytechnique Fédérale de Lausanne in der Schweiz. Scala ist unter anderem durch ihr Actor Framework populär geworden, welches sich stark an das Konzept aus Erlang anlehnt, und die nebenläufige Programmierung vereinfacht.

Alle folgenden Codebeispiele sind in Scala verfasst. Durch die höhere Ausdruckstärke gegenüber Java und die damit verbundene kompaktere Schreibweise, erleichtert Scala bei der Realisierung die Konzentration auf das Wesentliche.

Für die Nachbildung der Living-Place-Infrastruktur wurden ActiveMQ und MongoDB verwendet.

ActiveMQ<sup>6</sup> ist ein Message Bus, der eine lose Kopplung der Komponenten ermöglicht. Die Komponenten können über Kanäle (*Topics*) eine Multicast-Kommunikation aufbauen oder über Queues eine verlässliche Punkt-zu-Punkt-Kommunikation, in der Nachrichten so lange gespeichert werden, bis der Empfänger sie abholt.

MongoDB<sup>7</sup> ist eine dokumentenorientierte Datenbank, die aus dem NoSQL<sup>8</sup> Bereich stammt.

### 4.7. Systemaufbau

Die Abbildung 4.3 aus Abschnitt 4.2 gibt einen prinzipiellen Blick auf die gesamte Architektur des Living Place. Ergänzend dazu zeigt Abbildung 4.8 den Ausschnitt dieser Architektur, der in der vorliegenden Arbeit entwickelt wird. Dieser Ausschnitt zeigt hier entwickelten Komponenten Goal-Manager und ein Exemplar eines Goal-Agents, sowie die Infrastrukturkomponenten Message Bus und Message Store.

---

<sup>5</sup>Odersky u. a. (2008), <http://www.scala-lang.org>

<sup>6</sup><http://activemq.apache.org>

<sup>7</sup><http://www.mongodb.org>

<sup>8</sup>NoSQL steht für Not only SQL, eine Strömung für Datenbanken mit nicht-relationalen Ansatz, z. B. für MapReduce Anwendungen wie bei Googles BigTable.

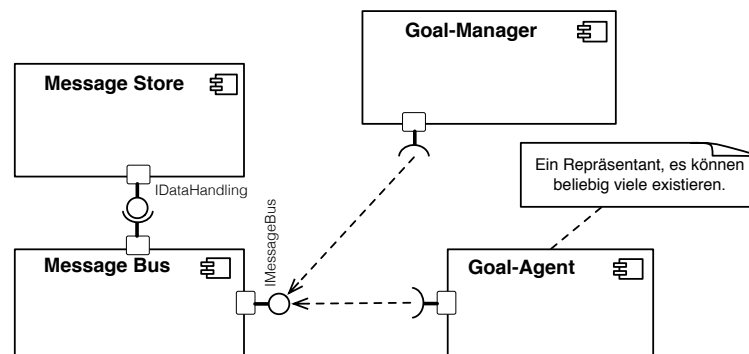


Abbildung 4.8.: Skizze der Komponenten

Der Goal-Manager und die Goal-Agents stellen Teilnehmer des Message Bus dar und agieren als reaktive oder proaktive Agenten. Sie sind über ihre Schnittstelle mit dem Message Bus verbunden und können über ihn Nachrichten senden und empfangen.

Als Message Bus kommt die ActiveMQ zur Verwendung. Dieser wird mit der Datenbank MongoDB kombiniert, um Nachrichten persistieren zu können, z. B. um eine Historie von Sensordatenverläufen zu speichern, die dann verknüpft und interpretiert werden können. Der Einsatz und die Arbeitsweise der ActiveMQ im Living Place der HAW Hamburg wird im Projektbericht von [Otto und Voskuhl \(2010\)](#) detailliert erläutert.

An den Message Bus angeschlossene Komponenten können verschiedene Kommunikationskanäle öffnen, über die sie in Kontakt treten. Diese Kanäle können themenbezogen sein. Sie bieten Multicast-Kommunikation an, bei der es z. B. einen Sender und  $n$ -Empfänger gibt. Das Einplanen von neuen Zielen ist solch ein Fall der Multicast-Kommunikation. Der Goal-Manager sendet ausschließlich, während die Goal-Agents ausschließlich empfangen.

Es werden drei Kanäle eingeführt:

- scenario activation** Zum Aktivieren oder Deaktivieren von Szenarien; Sender ist der Scenario-Reasoner; Empfänger ist der Goal-Manager.
- goal planning** Zum Einplanen neuer Ziele; Sender ist der Goal-Manager; Empfänger sind die Goal-Agents.
- goal coordination** Zur Koordination der Zielerfüllung; Sender und Empfänger sind die Goal-Agents.



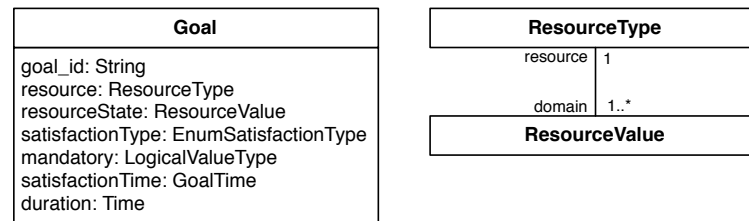


Abbildung 4.9.: UML-Modell von Zielen

## 4.8. Klassenmodell

Im folgenden Abschnitt wird das Klassenmodell des Prototypen entwickelt. Die Klassen sind **Goal** für Ziele, **GoalRelation** für Relationen zwischen den Zielen, **Scenario** für die Szenarios und **GoalManagementAgent** für den Goal-Manager.

### 4.8.1. Klasse Goal

Das UML-Modell in Abbildung 4.9 zeigt den Aufbau von Zielen. Ein **Goal** besitzt Informationen über die Zielresource (**resource**), deren Zustand (**resourceState**) und um was für einen Zieltypen es sich handelt (**satisfactionType**). Außerdem enthält es Informationen, ob es sich um ein obligatorisches Ziel handelt (**mandatory**) oder nicht, welche zeitlichen Grenzen das Ziel besitzt (**satisfactionTime**) und wie viel Zeit die Erfüllung des Ziel in Anspruch nimmt (**duration**). Eindeutig gekennzeichnet wird es durch die Identifikator des Ziels (**goal\_id**).

Der Parameter **resource** der Klasse **Goal** stellt einen Identifikator, eine URL oder einen ähnlichen Resourcelocator einer existierenden Ressource im System dar. Die Ressource kann z. B. ein Terminkalender, der Bewohner der Wohnung oder eine Kaffeemaschine sein. Der **resourceState** ist ein Wert aus der Wertedomäne der angegebenen Ressource. Mit dem Parameter **satisfactionType** wird dem Ziel einer der Zielerfüllungstypen aus Abschnitt 3.2.2 zugeordnet. Der Parameter **satisfactionTime** ist zum Zeitpunkt der Planung vermutlich nicht bekannt. In diesem Fall ist aber vielleicht der Parameter **duration** bekannt oder kann vom Goal-Agent vor der **satisfactionTime** ermittelt werden. Da zur Erfüllung eines Ziels und zur Erfüllung davon abhängender Ziele konkrete Zeiten notwendig sind, müssen diese miteinander in Beziehung gesetzt werden, wie in der Analyse beschrieben. Dieser Aspekt wird im Folgenden behandelt.

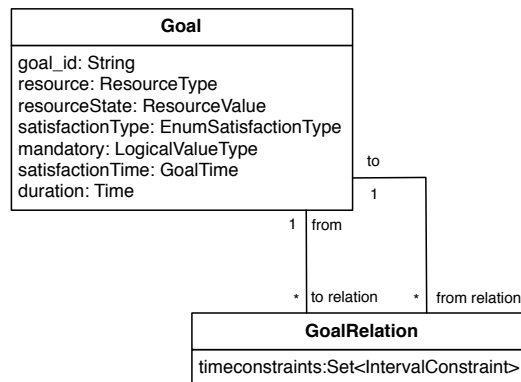


Abbildung 4.10.: Ziele können über eine Relationsklasse mit beliebig vielen Zielen in Relation stehen.

#### 4.8.2. Klasse GoalRelation

Damit die Ziele mit Hilfe von Allen-Intervall-Relationen relativ zueinander geordnet werden können, wird im UML-Diagramm in Abbildung 4.10 eine Klasse **GoalRelation** eingeführt. Ein Ziel kann dabei prinzipiell mit beliebig vielen anderen Zielen in Beziehung stehen.

Wie in der Abbildung 4.10 zu sehen, sind die UML-Assoziationsenden auf der **Goal**-Seite mit **to** und **from** bezeichnet. Da mit der Intervallalgebra immer die Relation **von** einem Intervall **zu** einem anderen angegeben wird, müssen auch die **GoalRelations** eine gerichtete Verbindung widerspiegeln. Die Klasse **GoalRelation** enthält zudem eine Menge von Intervallrelationen, die diese Verbindung gewichten.

Zur Laufzeit sollen Verbindungen zu einem Ziel, das **mandatory** ist, schwächer sein, als Verbindungen zu einem Ziel das nicht **mandatory** ist. Das soll implizit den obligatorischen Pfad ermöglichen, in dem nur Ziele vorhanden sind, die für das Szenario zwingend sind.

#### 4.8.3. Klasse Scenario

Unter Abschnitt 3.3 werden Szenarios als Graphen von Zielen aufgefasst. Die Modellierung der Szenarios zeigt Abbildung 4.11. Ein **Scenario** ist ein übergeordneter Container für die Klassen **Goal** und **GoalRelation**. Somit stellt die Klasse **Scenario** einen Graphen dar, so wie in Abschnitt 3.3 beschrieben.

Der folgende Scalacode<sup>9</sup> zeigt einen Ausschnitt aus der **Scenario**-Klasse:

```
1 class Scenario (relationMap : Map[Tuple2[Goal, Goal], GoalRelation]) {
```

<sup>9</sup>Unter Anhang B befindet sich eine auf diese Arbeit abgestimmte Kurzreferenz der Sprache Scala.

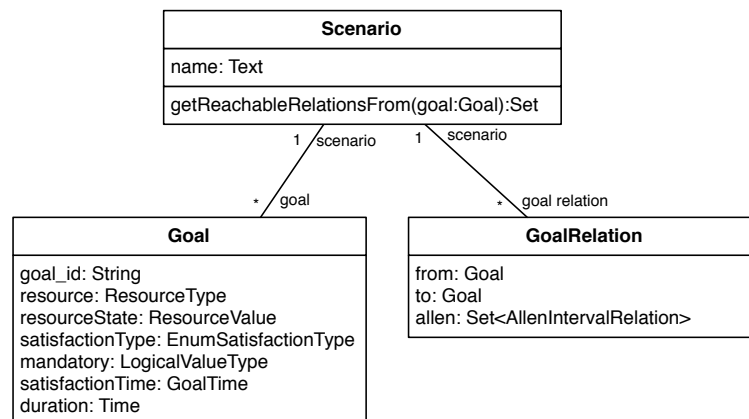


Abbildung 4.11.: UML-Modell von Szenarios.

```

2  val bindingOne = Set[Constraint.Value](b, m, o)
3  val bindingOnePlus = Set[Constraint.Value](o, m, fi)
4  // [...]
5  def getReachableRelationsFrom(goal: Goal): Set[GoalRelation] = {
6      return relationMap.collect {
7          case ((from, _), goalRelation)
8              if from == goal &&
9                 goalRelation.constraints_intersect(bindingOne).nonEmpty
10             => return goalRelation
11         case ((from, to), _)
12             if from != goal && reachable(goal, to, bindingOnePlus)
13             => return transitivity(goal, to)
14     }
15 }
16 }

```

Mit der Methode *getReachableRelationsFrom* sucht das **Scenario** anhand der Regeln aus Abschnitt 4.3 nach Folgezielen.

#### 4.8.4. Klassen **GoalManagementAgent** und **GoalManager**

Die Klasse **GoalManagementAgent** ist der Koordinator von verschiedenen Instanzen der Klasse **GoalManager**. Zudem empfängt und sendet er Nachrichten über den Message Bus und erstellt neue **GoalManager**, wenn ein neues Szenario aktiv wird. Außerdem verschickt der **GoalManagementAgent** Nachrichten, wenn neue Ziele geplant werden sollen.

Die Klasse **GoalManager** ermittelt die nächsten zu planenden Ziele, indem er ein bestimmtes **Scenario** untersucht. Im folgenden Scalacode wird der Algorithmus dazu dargestellt:

```
1 class GoalManager(scenario: Scenario) {
2   // [...]
3   def resolve(goal: Goal): Set[Goal] = {
4     var result = Set[Goal]()
5     var relations = scenario.getReachableRelationsFrom(goal)
6     relations.foreach { r: GoalRelation =>
7       result += r.to
8     }
9     return result
10  }
11
12  def getNextGoalsToBeReached(): Set[Goal] = {
13    if (!scenario.hasMarkedGoals()) {
14      return scenario.findStartingGoals
15    }
16
17    var reachable = Set[Goal]()
18    var marked: Set[Goal] = Set[Goal]()
19    marked += scenario.markedSolved
20    marked.foreach { goal =>
21      val next = resolve(goal)
22      reachable += next
23    }
24
25    reachable.collect { case goal if scenario.isNotSolved(goal) => goal }
26  }
27 }
```

Gilt ein Ziel als erfüllt, kann die *resolve* Methode die nächsten erreichbaren Zielen im Graphen auffinden. Sind diese nicht ihrerseits erfüllt, können sie als zu erfüllende Ziele eingeplant werden.

### 4.9. Kommunikation

Die unter Abschnitt 4.5 erwähnten Inter-Agent-Aspekte behandeln die technische Seite der Kommunikation. Diese werden in [Bedrouni u. a. \(2009\)](#) in zwei grundlegende Konzepte unterteilt:

**dezentral organisiert** Die Agenten lösen mit Hilfe von Verhandlungsstrategien Konflikte selbstständig. Diese Variante skaliert in der Regel gut, solange die Verhandlungsstrategien effizient arbeiten. Diese Organisation ist allerdings schwieriger zu implementieren als die zentrale Organisation.

**zentral organisiert** Dieses Konzept wird z. B. mit Hilfe eines Blackboards realisiert, das als Wächter über die Kommunikation fungiert, d. h. dass es die Agenten koordiniert. Diese Variante ist einfacher zu implementieren als eine dezentrale Organisation. Sie skaliert allerdings allgemein weniger gut, da sich das Blackboard als Nadelöhr erweisen kann.

Wie schon in Abschnitt 4.7 erwähnt stellen [Ellenberg u. a. \(2011\)](#) als zentrale Komponente des Living Places einen Message Bus, der gekoppelt mit einer Datenbank versendete Nachrichten speichern und damit als Blackboard verwendet werden kann. Je nach Anwendungsfall ist sowohl die dezentrale, als auch die zentrale Organisation umsetzbar. In der vorliegenden Arbeit wird eine dezentrale Organisation gewählt.

Unter der Vorgabe des Living Places einer Message-Driven-Architektur, können am Bus angeschlossene Komponenten folgendermaßen miteinander interagieren:

Zum einen können sie Nachrichten über frei wählbare Kanäle verschicken, zum anderen können sie Nachrichtenkanäle abonnieren, um alle Nachrichten, die über einen Kanal laufen, zu empfangen. Zudem können die Komponenten Punkt-zu-Punkt Kommunikation betreiben, indem eine Komponente die selbe Queue öffnet wie eine andere. Nachrichten werden im Living Place im JSON-Format versendet. JSON<sup>10</sup> ist ein simples, für den Menschen leicht les- und schreibbares Datenaustauschformat in Textform. Die Nachrichten werden als Key-Value-Map mit einem Zeitindex versehen in der Datenbank gespeichert. Somit ist es möglich die Nachrichten eines Kanals später wieder ablaufen zu lassen, z. B. um eine Folge von Nachrichten interpretieren zu können.

Im Folgenden werden die verschiedenen Nachrichtentypen, die für ein minimales Setting notwendig sind, erläutert.

##### 4.9.1. Szenario-Nachrichten

Der Scenario-Reasoner teilt dem **GoalManagementAgent** mit, ob er ein neues Szenario erkannt hat und welche Aktivitäten des Szenarios abgeschlossen sind. Aus Sicht des **GoalManagementAgents** sind diese Aktivitäten gleichbedeutend mit Zielen.

Die Nachrichten auf dem Kanal *scenario activation* haben folgende Syntax:

<b>scenarioid</b>	eindeutiger Identifikator des ausgewählten Szenarios
<b>action</b>	eine der beiden Aktionen <i>enter</i> oder <i>leave</i>

---

<sup>10</sup><http://www.json.org>

**satisfied\_activities** eine Liste von Identifikatoren abgeschlossener Aktivitäten des ausgewählten Szenarios

Ein Beispiel einer Nachricht im JSON-Format:

```
1 {
2   "scenarioId": "s1",
3   "action": "enter",
4   "satisfied_activities": "ac1,ac2"
5 }
```

#### 4.9.2. Goal-Management-Nachrichten

Für die einfachen Fälle der Koordination aus Abschnitt 4.5<sup>11</sup> können folgende Nachrichtentypen für das Planen von Zielen, das Fallenlassen von Zielen, falls in der Zwischenzeit eine Aktivität abgeschlossen wurde, und für das Zuweisen von Start- und Endzeit definiert werden.

Auf dem Nachrichtenkanal *goal planning* wird der Nachrichtentyp *goal\_planned* verschickt. Diese Nachricht ist analog zur Klasse **Goal** aufgebaut:

<b>goalId</b>	Ein eindeutiger Identifikator, damit Agenten bei ihrer Kommunikation ein Ziel referenzieren können.
<b>resourceId</b>	Ein Marker zum eindeutigen Identifizieren der Zielressource.
<b>resourceState</b>	Der Zielzustand, ein Wert aus der Wertedomäne der Ressource.
<b>mandatory</b>	Eine Makierung, ob ein Ziel obligatorisch ist.
<b>satisfactionType</b>	Zieltyp
<b>startTime</b>	Anfang des Zielintervalls.
<b>endTime</b>	Ende des Zielintervalls.
<b>allenRelationsToGoals</b>	Eine Menge mit allen ausgehenden Intervall-Relationen zu anderen Zielen.

Die Datentypen **GoalTime** und **GoalRelation** werden in der Nachricht aufgelöst in Start- und Endzeit, sowie in eine Liste, die die Intervall-Relationen des geplanten Ziels zu anderen Zielen enthält.

Ein Beispiel einer Nachricht im JSON-Format:

---

<sup>11</sup>Wenn z. B. darüber verhandelt werden muss, welcher Agent ein Ziel übernimmt, müssen weitere Nachrichtentypen definiert werden.

```
1 {
2   "goalId": "g1",
3   "resourceId": "r1",
4   "resourceState": "daytime",
5   "mandatory": "true",
6   "satisfactionType": "s1",
7   "startTime": "",
8   "endTime": "",
9   "duration": "",
10  "allRelationsToGoals": "g2_o,m,g3_b,m"
11 }
```

Ein Goal-Agent, der ein Ziel übernimmt, das keine ausgehenden Intervall-Relationen besitzt, muss keine anderen Ziele beachten. Im Umkehrschluss muss sich ein Goal-Agent, der ein Ziel übernimmt, das ausgehende Intervall-Relationen besitzt, mit anderen Goal-Agents koordinieren. Der zweite Nachrichtentyp *goal\_timeboundaries* dient dieser Koordination. Nachrichten diesen Typs werden über den Nachrichtenkanal *goal coordination* verschickt und weisen folgende Struktur auf:

**goalId** Der eindeutige Identifikator des geplanten Ziels.

**startTime** Die vom zuständigen Agenten ermittelte Startzeit des Ziels.

**endTime** Die vom zuständigen Agenten ermittelte Endzeit des Ziels.

Ein Beispiel einer Nachricht im JSON-Format:

```
1 {
2   "goalId": "g1",
3   "startTime": "06.45",
4   "endTime": "07.00",
5 }
```

Weiterhin kann der **GoalManagementAgent** den Goal-Agents mitteilen, ob ein **Goal** inzwischen erfüllt ist. Dabei handelt es sich um den Nachrichtentyp *goal\_solved*, der wie die erste Nachricht auf dem Kanal *goal planning* verschickt wird:

**goalId** Der eindeutige Identifikator des geplanten Ziels.

Ein Beispiel einer Nachricht im JSON-Format:

```
1 {
2   "goalId": "g1",
3 }
```

## 4.10. Tests und Ergebnisse der Simulation

Im Folgenden wird die Testumgebung sowie die mit dem Prototypen durchgeführten Tests erläutert. Anhand der Tests soll gezeigt werden, dass das Konzept tragfähig ist. Als erstes wird hierzu die Funktion der Klasse **GoalManager** dargestellt und dessen Funktionsweise anhand der in Kapitel 3 erarbeiteten Szenarios demonstriert. Das spätere Zusammenspiel im Living Place wird durch einen Versuchsaufbau mit einem **GoalManagementAgent** und mehreren Goal-Agents gezeigt.

### 4.10.1. GoalManager

Um den **GoalManager** zu testen werden die Userstories aus Abschnitt 3.1 kodiert. Am Anfang des Beispiels wird Userstory im Code als Scenario definiert. Dazu werden die Klassen **Goal**, **GoalRelation** und **Scenario** verwendet.

Die drei Klassen werden folgendermaßen definiert:

```
1 case class Goal(goal_id: String, resource: String, resourceState: String,
2                 mandatory: Boolean, satisfactionType: GoalType)
3
4 case class GoalRelation (fromGoal: Goal,
5                          constraints: Set[Constraint.Value],
6                          toGoal: Goal)
7
8 class Scenario(relations: Set[GoalRelation])
```

Die Klasse **Goal** wird über die Parameter aus Unterabschnitt 4.8.1, abzüglich **duration** und **satisfactionTime** definiert. Die drei Parameter **goal\_id**, **resource** und **resourceState** müssen eindeutig sein. Der Parameter **mandatory** legt fest, ob das Ziel zum obligatorischen Pfad gehört. Der Parameter **satisfactionType** entspricht einem Symbol des Aufzählungstyps **GoalType**: **FROM\_START**, **EXACTLY\_TO\_END** und **UNTIL\_END**. Die drei Symbole entsprechen den drei Zieltypen aus Abbildung 3.1.

### Tagesbeginn-Szenario

Das Tagesbeginn-Szenario entspricht prinzipiell der in der Analyse skizzierten Userstory Tagesbeginn (Abbildung 3.3). Es wird folgendermaßen kodiert:

```
1 // Definition eines Goals / Reihenfolge der Parameter:
2 // goal_id, resource, resource_state, mandatory, satisfaction_type
3
4 val g1 = new Goal("g1", "heater", "at_daymode", true, EXACTLY_TO_END)
5 val g2 = new Goal("g2", "human", "is_aware", true, EXACTLY_TO_END)
```



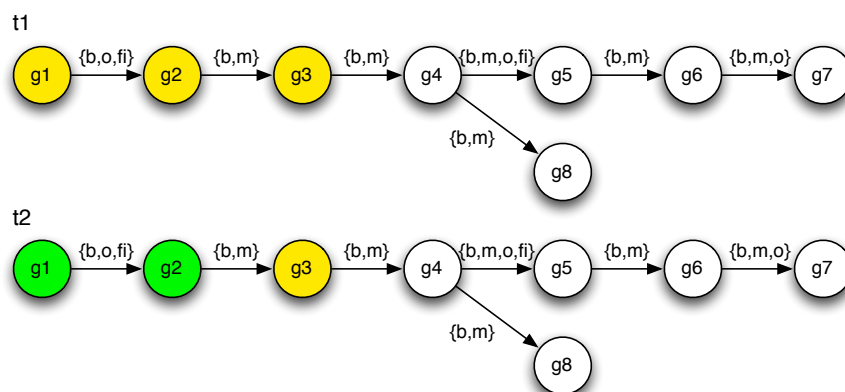
#### 4. Design und Realisierung

```
6 val g3 = new Goal("g3", "human", "taken_a_shower", true, EXACTLY_TO_END)
7 val g4 = new Goal("g4", "human", "is_dressed", true, UNTIL_END)
8 val g5 = new Goal("g5", "coffee_machine", "is_cooked", true, UNTIL_END)
9 val g6 = new Goal("g6", "human", "had_breakfast", true, UNTIL_END)
10 val g7 = new Goal("g7", "human", "is_informed_about_day's_schedule", true, UNTIL_END)
11 val g8 = new Goal("g8", "documents", "are_provided", true, FROM_START)
```

Die soeben definierten Ziele werden anschließend mit Relationen verbunden, so dass aus den Zielen eine partiell zeitlich geordnete Menge an Zielen wird. Aus dieser Ordnung besteht das eigentliche Szenario. Mit Hilfe der Klassen **GoalRelation** und **Scenario** wird diese Struktur intern als Graph dargestellt:

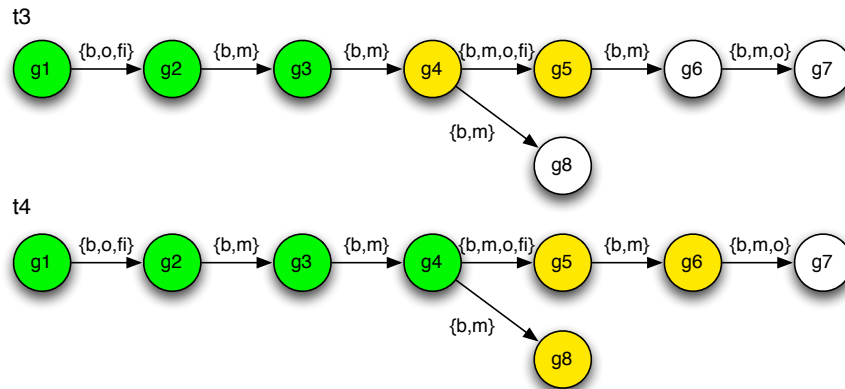
```
1 val gr1 = GoalRelation(g1, Set(b, o, fi), g2)
2 val gr2 = GoalRelation(g2, Set(b, m), g3)
3 val gr3 = GoalRelation(g3, Set(b, m), g4)
4 val gr4 = GoalRelation(g4, Set(b, m, o, fi), g5)
5 val gr5 = GoalRelation(g5, Set(b, m), g6)
6 val gr6 = GoalRelation(g6, Set(b, m, o), g7)
7 val gr7 = GoalRelation(g4, Set(b, m), g8)
8 val goalRelations = Set[GoalRelation](gr1, gr2, gr3, gr4, gr5, gr6, gr7)
9
10 val scenario = new Scenario(goalRelations)
```

Der **GoalManager** ist mit diesem Szenario initialisiert. Noch haben keine Aktivitäten stattgefunden, das heißt, keines der Ziele ist bisher erfüllt worden. Daher wählt der **GoalManager** bei  $t_1$  die Ziele  $g_1$  und  $g_2$  aus und propagiert diese an die Goal-Agents. Das bedeutet, es wird erwartet, dass die Heizung die Wohnung auf Tagestemperatur aufheizt und Sal jr. geweckt wird. Die Gewichtung zwischen  $g_1$  und  $g_2$  enthält die Relation  $fi$ , die Ziele dürfen dadurch auch zeitgleich erfüllt werden. Aus diesem Grund sind zum Zeitpunkt  $t_2$  beide Ziele erfüllt. Die Relationen  $b, m$  zwischen  $g_2$  und  $g_3$  führen dazu, dass in  $t_2$  nur  $g_3$  als Folgeziel ausgewählt wird, das heißt, Sal jr. kann nur duschen, wenn sie vorher geweckt wurde.

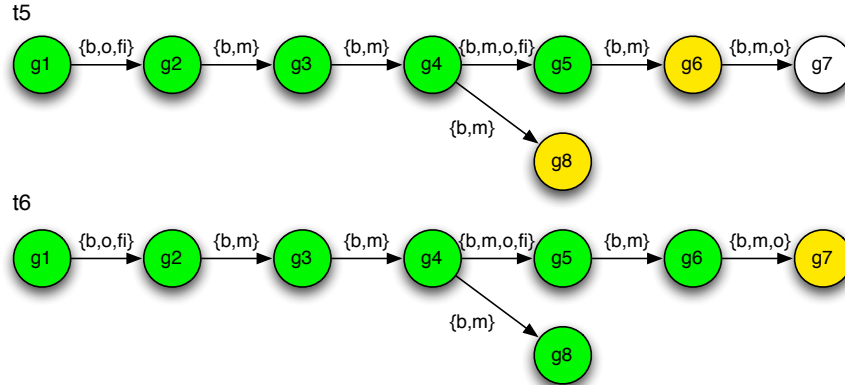


#### 4. Design und Realisierung

Zum Zeitpunkt  $t_3$  ist auch Ziel  $g_3$  erfüllt. Daraufhin wählt der **GoalManager** das Folgeziel  $g_4$  aus, das über die Relationen  $b, m$  erreichbar ist. Anders sieht es zum Zeitpunkt  $t_4$  aus. Das Ziel  $g_4$  ist erfüllt. Da die Gewichtung zwischen den Ziele  $g_4$  und  $g_5$  die Relation  $fi$  enthält, ergibt sich eine transitive Gewichtung  $g_4 \rightarrow g_6 = \{b, m\}$ . Die Relation  $m$  führt nach den Regeln aus Abschnitt 4.3.4 dazu, dass auch  $g_6$  auf  $g_4$  folgt. Somit werden korrekterweise auf dem einen Pfad  $g_5$  und  $g_6$  als Folgeziele markiert und auf dem anderen Pfad  $g_8$ .



Sobald  $g_6$  als erfüllt markiert wird, wird  $g_7$  eingeplant.



Sal jr.'s Tagesbeginn-Szenario ist damit vollständig abgelaufen. Das bedeutet, der **GoalManager** arbeitet wie erwartet. Dass das Szenario linear abläuft ist auf die spezielle Struktur dieses Szenarios und auf die gewählte Abfolge der erfüllten Ziele zurückzuführen. Die Reihenfolge der Ziele wird außerdem nicht über die Kanten bestimmt, sondern einzig über die zwischen den Knoten bestehende zeitliche Ordnung, die über die Zeitlogik von Allen definiert wird.

### Besuch von Freunden-Szenario

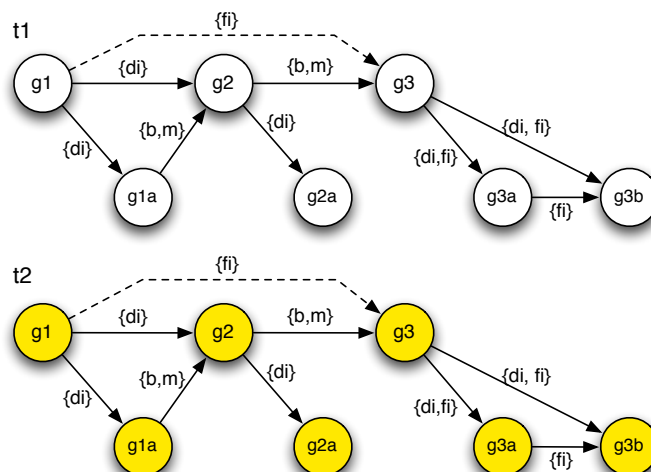
Das zweite Szenario entspricht prinzipiell der in der Analyse skizzierten Userstory: Besuch von Freunden (Abbildung 3.4). Es wird folgendermaßen kodiert:

```

1 // Definition eines Goals / Reihenfolge der Parameter:
2 // goal_id, resource, resource_state, mandatory, satisfaction_type
3 val g1 = new Goal("g1", "calendar", "visitation_appointment_inscribed",
4                                     true, FROM_START)
5 val g1a = new Goal("g1a", "calendar", "appointment_type_assigned", false, UNTIL_END)
6 val g2 = new Goal("g2", "human", "chosen_meal", true, UNTIL_END)
7 val g2a = new Goal("g2a", "homemaker", "list_of_meals_composed", false, FROM_START)
8 val g3 = new Goal("g3", "homemaker", "stock_of_food_assured", true, EXACTLY_TO_END)
9 val g3a = new Goal("g3a", "homemaker", "shopping_list", false, FROM_START)
10 val g3b = new Goal("g3b", "human", "shopping_is_done", false, UNTIL_END)
11
12 val gr1 = GoalRelation(g1, Set(di), g1a).withAddingDependency
13 val gr2 = GoalRelation(g1a, Set(b, m), g2).withAddingDependency
14 val gr3 = GoalRelation(g2, Set(di), g2a).withAddingDependency
15 val gr4 = GoalRelation(g2, Set(b, m), g3).withAddingDependency
16 val gr5 = GoalRelation(g3, Set(di, fi), g3a).withAddingDependency
17 val gr6 = GoalRelation(g3, Set(di, fi), g3b).withAddingDependency
18 val gr7 = GoalRelation(g3a, Set(fi), g3b).withAddingDependency

```

Zum Zeitpunkt  $t1$  hat der **GoalManager** noch keine Ziele ausgewählt und keines der Ziele ist erfüllt. Die gestrichelt dargestellte Kante hätte analog zur Abbildung 3.4 ebenso definiert werden können, da ihr Vorhanden- oder Nichtvorhandensein den Versuchsablauf aber nicht beeinflusst, wurde diese aus Gründen der Vereinfachung nicht berücksichtigt. Das Verhalten des **GoalManagers** ändert sich dadurch nicht. Die Hauptziele sind  $g1$  (Termin im Kalender),  $g2$  (Auswahl eines Gerichtes) und  $g3$  (ausreichend Lebensmittel vorhanden). Die anderen Ziele sind Hilfsziele. Sie werden im **GoalManager** nicht gesondert behandelt.



Zum Zeitpunkt  $t_2$  wählt der **GoalManager** die ersten zu erfüllenden Ziele aus. Da  $g_1$  prinzipiell parallel zu allen anderen Zielen existiert, werden in diesem Szenario alle Ziele im ersten Schritt eingeplant.

Das Szenario Besuch von Freunden ist damit vollständig geplant. Der **GoalManager** kann keine weiteren Ziele mehr einplanen. Die Erfüllung des Szenarios hängt jetzt davon ab, ob die Ziele erreicht werden. Der **GoalManager** arbeitet wie erwartet.

#### 4.10.2. Ergebnisse der Simulation

Um einen ersten Eindruck zu bekommen, wie sich der **GoalManager** in einer praxisnahen Umgebung verhält, ist ein Versuch mit einer isolierten Infrastruktur aufgebaut worden. Auf einem lokalen Rechner wurde die Softwareumgebung mit ActiveMQ und MongoDB installiert. Der **GoalManager** und die für diesen Versuch implementierten Goal-Agents benutzen den für Living-Place-Anwendungen bereitgestellten Living-Place-Adapter.

Die Nachrichten des Scenario-Reasoners, wie *scenario\_selected* oder *scenario\_goal\_solved*, werden von Hand mittels einer Testapplikation gesendet. Der **GoalManagerAgent** verarbeitet diese analog zum vorherigen Funktionstest und wird die Nachrichten GOAL\_PLANNED und GOAL\_SOLVED an die Goal-Agents versenden. Die Goal-Agents reagieren auf die Nachrichten GOAL\_PLANNED, GOAL\_TIMEBOUNDARIES und GOAL\_SOLVED folgendermaßen:

- GOAL\_PLANNED** Der Agent übernimmt ein Goal, wenn er die passende Ressource zum Goal verwaltet. In diesem Versuchsaufbau verwaltet der Agent in einigen Fällen die Ressource nur für einen bestimmten Zielzustand. Hat der Agent ein Goal übernommen, wird er weitere GOAL\_PLANNED Nachrichten verwerfen.
- GOAL\_TIMEBOUNDARIES** Aktualisiert ein Goal sein Zeitintervall, so prüft der Agent, ob sein Goal von diesem abhängig ist, wenn ja, aktualisiert er das Zeitintervall seines Ziels und erfüllt dieses anschließend. In diesem Versuchsaufbau wird der Agent dazu ein vordefiniertes Intervall auf das Ziel, das er verwaltet, und verschickt die Nachricht GOAL\_TIMEBOUNDARIES.
- GOAL\_SOLVED** Ist das Goal, das der Agent übernommen hat, erfüllt, so löscht er das Goal aus seinem Speicher und wartet auf eine neue Nachricht GOAL\_PLANNED.

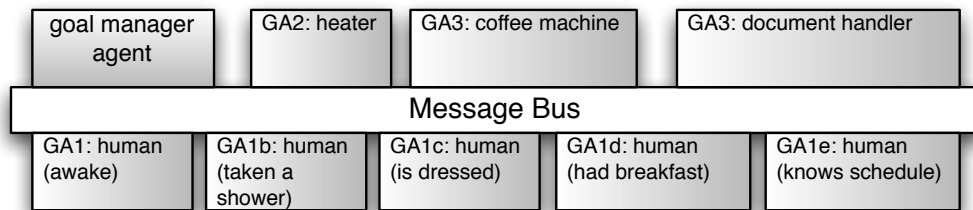


Abbildung 4.12.: Versuchsaufbau Tagesbeginn-Szenario

### Tagesbeginn-Szenario

In diesem Versuch wird wieder das Tagesbeginn-Szenario aus dem **GoalManager**-Test genommen. Zu beachten ist an dieser Stelle die Information **mandatory**. Diese ist für alle Ziele auf wahr gesetzt. Damit müssen alle Ziele des Szenarios erfüllt werden, um das Szenario nicht zu verlassen.

Der **GoalManager** wird mit dem gleichen kodierten Tagesbeginn-Szenario wie im **GoalManager**-Test initialisiert.

Es stehen acht Goal-Agents zur Verfügung. Da es keinen real existierenden Bewohner im Test gibt, verwalten fünf Agenten Ziele, die den Bewohner als Ressource angegeben haben. Es sind fünf Agenten, die jeweils für einen Zielzustand zuständig sind, um Fehlerquellen zu vermeiden. Die einzelnen Agenten bleiben so in ihrem Aufbau sehr einfach. Die weiteren drei Agenten verwalten Ziele mit den Ressourcen: heater, coffee machine und document handler. Beispiele für die Implementierung der Goal-Agents sind im Anhang [D.2](#) zu finden.

Die folgende Tabelle zeigt einen aufbereiteten Ausschnitt des Console-Logs aus dem Anhang [D.3.1](#). Es zeigt welche Nachrichten von welchem Agenten abgegeben wurden und welcher Agent Nachrichten verarbeitet hat. Die Abkürzung SI steht für Scenario-Reasoner, GM für GoalManager, GA jeweils für einen Goal-Agent.  $GA_1$  bis  $GA_{1e}$  behandeln die Ziele für den Menschen,  $GA_2$  ist für die Heizung zuständig,  $GA_3$  für die Kaffeemaschine und  $GA_4$  für den

Dokumenten-Verwalter. Der Zeitpunkt ( $t_i$ ) orientiert sich an der Zeit, wann der Konsument die Nachricht empfangen hat. Sie kann zu einem früheren Zeitpunkt abgeschickt worden sein.

Zeit:	Sender	Nachricht	Konsument
		<i>SCENARIO_SELECTED</i>	
$t_1$ :	<i>SI</i>	→	<i>GM</i>
		<i>GOAL_PLANNED(g<sub>1</sub>)</i>	
$t_2$ :	<i>GM</i>	→	<i>GA<sub>2</sub></i>
		<i>GOAL_PLANNED(g<sub>2</sub>)</i>	
$t_3$ :	<i>GM</i>	→	<i>GA<sub>1</sub></i>
		<i>GOAL_PLANNED(g<sub>3</sub>)</i>	
$t_4$ :	<i>GM</i>	→	<i>GA<sub>1b</sub></i>
		<i>GOAL_TIMEBOUNDARIES(g<sub>1</sub>)</i>	
$t_5$ :	<i>GA<sub>2</sub></i>	→	<i>GA<sub>1</sub></i>
		<i>GOAL_TIMEBOUNDARIES(g<sub>2</sub>)</i>	
$t_6$ :	<i>GA<sub>1</sub></i>	→	<i>GA<sub>1b</sub></i>
		<i>SCENARIO_GOAL_SOLVED(g<sub>1</sub>)</i>	
$t_7$ :	<i>SI</i>	→	<i>GM</i>
		<i>SCENARIO_GOAL_SOLVED(g<sub>1</sub>)</i>	
$t_8$ :	<i>SI</i>	→	<i>GM</i>
		<i>GOAL_SOLVED(g<sub>1</sub>)</i>	
$t_9$ :	<i>GM</i>	→	<i>GA<sub>2</sub></i>
		<i>SCENARIO_GOAL_SOLVED(g<sub>2</sub>)</i>	
$t_{10}$ :	<i>SI</i>	→	<i>GM</i>
		<i>GOAL_SOLVED(g<sub>2</sub>)</i>	
$t_{11}$ :	<i>GM</i>	→	<i>GA<sub>1</sub></i>
		<i>SCENARIO_GOAL_SOLVED(g<sub>3</sub>)</i>	
$t_{12}$ :	<i>SI</i>	→	<i>GM</i>
		<i>GOAL_PLANNED(g<sub>5</sub>)</i>	
$t_{13}$ :	<i>GM</i>	→	<i>GA<sub>3</sub></i>
		<i>GOAL_SOLVED(g<sub>3</sub>)</i>	
$t_{14}$ :	<i>GM</i>	→	<i>GA<sub>1b</sub></i>
		<i>GOAL_PLANNED(g<sub>4</sub>)</i>	
$t_{15}$ :	<i>GM</i>	→	<i>GA<sub>1c</sub></i>

Wie im Ablauf des Tests zu erkennen ist, verhält sich die Testumgebung, wie es nach dem Test des **GoalManagers** zu erwarten war. Die Testumgebung bildet prinzipiell die in der Analyse beschriebene Userstory 1 ab. Alle Ziele können von ihren jeweiligen Agenten abgearbeitet werden.

### Besuch von Freunden-Szenario

Für diesen Test wird das kodierte Besuch von Freunden-Szenario aus dem **GoalManager**-Test verwendet. In diesem Szenario werden Hilfsziele definiert, die nicht **mandatory** sind. Dazu zählen die Ziele  $g1a$ ,  $g2a$ ,  $g3a$  und  $g3b$ . Diese Ziele müssen nicht mit eingeplant werden. Selbst wenn diese nicht erfüllt werden, besteht das Szenario weiter.

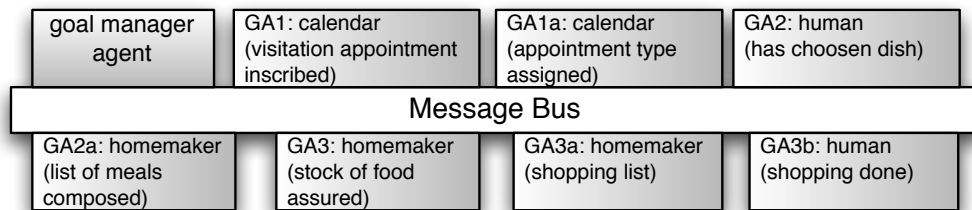


Abbildung 4.13.: Versuchsaufbau Besuch von Freunden-Szenario

Für dieses Szenario werden sieben Goal-Agents implementiert. Die oben genannten Hilfs- oder Subziele, werden in der Laufzeit zwar von **GoalManager** geplant, aktiv werden sollen sie aber erst, wenn sie durch den Goal-Agent, der das Hauptziel übernimmt, extra als Hilfsziel eingeplant werden.

Bei der Implementierung dieses Szenarios zeigt sich, dass die Hilfsziele eine besondere Behandlung benötigen. Der **GoalManager** plant die Hilfsziele im ersten Schritt normal ein. Ob sie jedoch benötigt werden, entscheidet sich erst zur Laufzeit der Goal-Agents. Daher wird der Nachrichtentyp `SUBGOAL_PLANNED` eingeführt. Ein Agent, der ein Hilfsziel übernimmt, fängt erst an dieses abzuarbeiten, wenn es vom Goal-Agent, der das Hauptziel übernommen hat, über diese Nachricht angefordert wird.

Wie im Versuchsaufbau des Tagesbeginn-Szenarios zeigt die folgende Tabelle einen Ausschnitt des Console-Logs (Anhang [D.3.2](#)).  $GA_1$  verarbeitet das Ziel `visitation_appointment_inscribed` für den Kalender,  $GA_{1a}$  verarbeitet das Ziel `appointment_type_assigned` für den Kalender,  $GA_2$  verarbeitet das Ziel `choosen_meal` für den Menschen,  $GA_{2a}$  verarbeitet das Ziel `list_of_meals_composed` für den Haushälter (homemaker),  $GA_3$  verarbeitet das Ziel `stock_of_food_assured` für den Haus-

halter,  $GA_{3a}$  verarbeitet das Ziel *shopping\_list* fur den homemaker und  $GA_{3b}$  verarbeitet das Ziel *shopping\_done* fur den Menschen.

Zeit:	Sender	Nachricht	Konsument
		<i>SCENARIO_SELECTED</i>	
$t_1$ :	$SI$	→	$GM$
$t_2$ :	$GM$	<i>GOAL_PLANNED(g<sub>3b</sub>)</i>	$GA_{3b}$
$t_3$ :	$GM$	<i>GOAL_PLANNED(g<sub>2a</sub>)</i>	$GA_{2a}$
$t_4$ :	$GM$	<i>GOAL_PLANNED(g<sub>2</sub>)</i>	$GA_2$
$t_5$ :	$GM$	<i>GOAL_PLANNED(g<sub>1a</sub>)</i>	$GA_{1a}$
$t_6$ :	$GM$	<i>GOAL_PLANNED(g<sub>3</sub>)</i>	$GA_3$
$t_7$ :	$GM$	<i>GOAL_PLANNED(g<sub>3a</sub>)</i>	$GA_{3a}$
$t_8$ :	$GM$	<i>GOAL_PLANNED(g<sub>1</sub>)</i>	$GA_1$

Mit dem abgebildeten Ablauf ist die initiale Phase abgeschlossen, alle Ziele sind vom **Goal-Manager** geplant und alle Goal-Agents haben jeweils ein Ziel konsumiert. In der anschlieenden Phase (ab  $t_9$ ) richten sich die Goal-Agents mit ihren Ziele zeitlich aus und die Hilfsziele (*SUBGOAL\_PLANNED*) werden von den Goal-Agents der Hauptziele geplant.

Zeit:	Sender	Nachricht	Konsument
		<i>SUBGOAL_PLANNED(g<sub>1a</sub>)</i>	
$t_9$ :	$GA_1$	→	$GA_{1a}$
$t_{10}$ :	$GA_1$	<i>GOAL_TIMEBOUNDARIES(g<sub>1</sub>)</i>	$GA_{1a}$
$t_{11}$ :	$GA_{1a}$	<i>GOAL_TIMEBOUNDARIES(g<sub>1a</sub>)</i>	$GA_2$
$t_{12}$ :	$GA_2$	<i>GOAL_TIMEBOUNDARIES(g<sub>2</sub>)</i>	$GA_3$
$t_{13}$ :	$GA_2$	<i>SUBGOAL_PLANNED(g<sub>2a</sub>)</i>	$GA_{2a}$

Wurden durch die Goal-Agents der Hauptziele auch die Hilfsziele geplant, konnen die Goal-Agents der Hilfsziele diese zeitlich ausrichten. Im Anschluss daran konnen die von den Hilfszielen abhangigen Ziele ebenfalls zeitlich ausgerichtet werden.

Der Versuch zeigt, dass sich das Szenario wie gewunscht aufspannt und alle Agenten ihre Ziele abarbeiten konnen. Das Console-Log zu diesem Test ist im Anhang [D.3.1](#) einzusehen.



### 4.11. Erkenntnisse aus Realisierung und Tests

Der **GoalManager** arbeitet mit einer Menge von Zielen, die er in drei Kategorien einteilt: Die unerfüllten Ziele, die zur Erfüllung ausgewählten Ziele und die erfüllten Ziele. Alle unerfüllten Ziele sind potentielle Ziele, die später eingeplant werden können. Die zur Erfüllung ausgewählten Ziele werden den Agenten mitgeteilt. Die erfüllten Ziele werden von der Planung ausgenommen bzw. den Agenten als erfüllt gemeldet.

Mit den Versuchsaufbauten wurde belegt, dass aus einem Szenario-Graphen die jeweils aktuellen Folgeziele ausgewählt werden können. Zudem wird gezeigt, dass die Koordination einer Agentengemeinschaft, in der jeder Agent einen Ressourcentyp vertritt, keine großen Anforderungen stellt. Die Agenten organisieren die Erfüllung ihrer Ziele zeitlich gemäß der Allen-Relationen, die zwischen den Zielen bestehen. Mit den Versuchen ist gezeigt, dass das hier entwickelte Konzept im Rahmen des prototypischen Aufbaus funktioniert und die aufgestellten Anforderungen erfüllt.

### 4.12. Evaluation des Designs und der Realisierung

Das hier entwickelte Design ermöglicht es komplexe und einen größeren Zeitraum überspannende Lebensabläufe in einem Smart-Home relativ simpel abzubilden. Die Zielesemantik verbirgt die technische Komplexität der Aufgaben hinter den Zielen. Die zentral hinterlegte, geräteunabhängige und vor allem zusammenhängende Beschreibung der Szenarios durch einer Struktur von Zielen, die über die Intervallgebra von Allen in eine relative zeitliche Ordnung gebracht werden, lässt eine einfachere Pflege dieser Beschreibung zu, als dies durch eine verteilte Struktur möglich wäre. In Zusammenarbeit mit einem Scenario-Reasoner steckt viel Potential im Design des Goal-Managers.

Die in den Grundlagen vorgestellten Smart-Home-Projekte konnten mit den bisherigen Lösungen lediglich direkte Anforderungen abdecken. Dieses System kann die Projekte in der Hinsicht erweitern, dass es eine vorausschauende Unterstützung für die Bewohner anbietet und in der Lage ist einzelne Situationen als größeren Zusammenhang zu betrachten und diesbezüglich auszuwerten.

Auch wenn in den Beispielen die Abarbeitung der Ziele deterministisch wirkt, ist sie abhängig von der tatsächlich geglückten Erfüllung der Ziele, sowie dem dynamischen Kontext der Wohnung. Dies unterscheidet sich stark von Geschäftsszenarien, in denen meist ein strenger Workflow vorgegeben und in der Regel auch notwendig ist, z. B. um die Revisionssicherheit zu gewährleisten. Bei Szenarien im Alltag können die einzelnen Abläufe stark variieren und dennoch die gleiche Lebenssituation widerspiegeln. Dies wird ansatzweise mit dem Szenario 1 gezeigt, in dem mehrere

Ziele zeitlich parallel liegen können ( $\{g1, g2\}$  und  $\{g4, g8, g5\}$ ) und deren Erfüllungsreihenfolge unabhängig von der Gesamterfüllung des Szenarios ist.

Die hier durchgeführten Versuche können nur unzureichend die Störanfälligkeit und Ausfallsicherheit eines realen Systems überprüfen. Eine genauere Untersuchung z. B. auf Deadlock- und Livelock-Eigenschaften wird daher angeraten<sup>12</sup>. Genauer ist zu überprüfen, ob Situationen existieren, in denen Goal-Agents nie terminieren, weil sie entweder nicht die nötigen Informationen beschaffen können, um ihr Ziel zu erfüllen, oder die Zielerfüllung nicht einleiten können, weil ihnen Zeitgrenzen anderer Ziele fehlen.

Zudem bleibt die Frage nach dem Umgang mit Hilfszielen. In den vorgestellten Versuchen werden die Hilfsziele stets eingeplant. Ob die Umgebung korrekt funktionieren wird, wenn Hilfsziele verworfen, weil nicht benötigt, werden, ist mit diesem ersten Prototypen nicht festzustellen.

Ein weiteres Modellierungsproblem stellt sich im Zusammenhang mit Kausalität. Zwar wird im Prototypen über **mandatory** ein obligatorischer Pfad vorgegeben, doch wird damit nicht die Kausalität zwischen den Zielen festgelegt. Ob und inwieweit Kausalität abgebildet werden muss und kann, ist in diesem Zusammenhang interessant.

Das Übersetzen einer Userstory in ein formales Szenario ist komplex und nicht trivial. Es wäre daher sinnvoll ein Tool zu entwickeln, welches das Erstellen von Szenarios unterstützt.

Es muss weiter untersucht werden, wie sich das System verhält, wenn mehrere Agenten ein Ziel erfüllen können, und wie komplexere Ziele abgebildet und gelöst werden können. In einem nächsten Schritt ist ein reales Szenario mit echten Goal-Agents zu erschaffen, um die Funktionsweise des hier erarbeiteten Konzepts zu bestätigen und Verhandlungsstrategien zwischen den Goal-Agents zu entwickeln.

Ein anderer interessanter Aspekt ist das Planen von Alternativzielen oder die Behandlung von Alternativpfaden vom **GoalManager**.

---

<sup>12</sup>Für diese Art von Eigenschaften bieten sich z. B. Petrinetze an, siehe dazu [Gottmann und Nachtigall \(2011\)](#) oder auch [Padberg \(1992 – 2010\)](#).

## 5. Zusammenfassung und Ausblick

### 5.1. Zusammenfassung

In den Grundlagenkapiteln wurden aktuelle Smart-Home-Projekte vorgestellt und deren Vorgehensweise aufgrund ihrer Abstraktionsebenen der Kontextverarbeitung kategorisiert. Die vorgestellten Smart-Home-Projekte lösen eher die technischen Aspekte der Sensorverarbeitung, der Verfügbarkeit und dem Auffinden von Diensten, in dieser Arbeit als Ebenen eins und zwei gekennzeichnet. Hingegen beschäftigt sich das CoFriend-Projekt mit der dritten Ebene, der Behaviour-Interpretation. In diesem Projekt wurde die korrekte Abfertigung eines Flugzeugs am Terminal erfolgreich erkannt. Die Behaviour-Interpretation und besonders die Behaviour-Prediction ist ein Bereich, der in Smart-Homes noch unzureichend erforscht ist.

In der Analyse wurde anhand von zwei Userstories gezeigt, wie Wohnsituationen als so genannte Szenarios betrachtet werden können, deren einzelne Aktivitäten als Ziele formuliert werden. Selbst komplexe Aktivitäten lassen sich auf diese Weise simpel darstellen. Die Ziele wurden als Zeitintervalle betrachtet und, bezogen auf den Erfüllungszeitpunkt in diesem Intervall, einem von drei Typen zugeordnet. Zum ersten Typ zählen Ziele, deren Erfüllung vom Beginn an bis zum Ende des Zeitintervalls gelten muss, zum zweiten gehören Ziele, deren Erfüllung genau am Ende des Intervalls eintritt, und zum dritten Typ zählen Ziele, deren Erfüllung spätestens zum Ende eintreten soll. Mit Hilfe dieser Zieldefinition konnten Szenarios aufgebaut werden. Dazu wurden die Ziele mit der Intervallalgebra von Allen in Relation und in eine relative zeitliche Ordnung gebracht. Es wurde festgestellt, dass Szenarios aus obligatorischen und optionalen Zielen bestehen. Obligatorische Ziele müssen erfüllt werden, sonst entspricht das Szenario nicht der Lebenssituation. Die optionalen Ziele können dagegen wegfallen, das Szenario ist in dem Fall dennoch weiterhin gültig. Die Szenarios sollen als eine zentrale Beschreibung von Lebenssituationen dienen, die unabhängig von den gerade tatsächlich im System befindlichen Agenten existieren können. Das soll die Flexibilität des Systems erhöhen. Zudem soll das Planen von Zielen Probleme des klassischen Planens vermeiden.

Mit Hilfe dieser Erkenntnisse wurde in Kapitel Design und Realisierung ein Modell eines prototypischen Goal-Managers entwickelt, der diese Szenarios kennen soll und anhand des

aktuellen Kontextes Ziele an Goal-Agents weitergibt. Der Goal-Manager löst das Reasoning über die Intervallalgebra und damit das Auffinden von Folgezielen, indem er die Szenarios als Graphen betrachtet, deren Kanten mit Intervallrelationen gewichtet werden. Zur konsistenten Verarbeitung der Intervallrelationen, wurde ein kleines Subset der Intervallalgebra definiert, das auf Grundlage der Konvexitätsbedingung in Bezug auf die Algebra konsistent ist.

Anschließend wurde ein Prototyp implementiert, der aus einem Goal-Manager und verschiedenen Goal-Agents besteht. In einer lokalen Testumgebung wurden zwei Szenarios auf Grundlage der beiden bekannten Userstories kodiert. Anhand dieses Aufbaus konnte die generelle Funktionsweise des Konzepts bestätigt werden. Das Konzept war mit relativ einfachen Mitteln zu realisieren. Die Versuche zeigten, dass der Goal-Manager Folgeziele korrekt auswählt und eine Menge von Goal-Agents sich automatisch aufgrund der zwischen ihren Zielen bestehenden Relationen zeitlich ausrichtet.

### 5.2. Ausblick

Es wäre wünschenswert, dass die Technik die Lebenssituationen inklusive der emotionalen Bedeutung erkennt und anschließend adäquat auf den Menschen eingeht, um nicht nur den Menschen besser unterstützen zu können, sondern auch die Akzeptanz dieser Technologien zu erhöhen. Die Computertechnik verschmilzt mit der Umwelt, so dass es ganz natürlich ist, mit ihr umzugehen und von ihr unterstützt zu werden.

Neben der eher passiven Unterstützung, durch z. B. Erinnerungen und Nachfragen, wären aktive persönliche Assistenten des Menschen wünschenswert, die gezielt Aufträge erfüllen können. Als Beispiel sei hier ein Kaufagent genannt, der selbstständig ein zu Wunscheigenschaften passendes Produkt findet und kauft.

Es gibt im Smart-Home-Forschungszweig und dem Gesamtbereich der Companion-Technology noch viele offene Fragen auf dem Weg zur Verwirklichung der geschilderten Vision.

## Literaturverzeichnis

- [sfbtransregio:62] : *Companion Technology*. – URL <http://www.sfb-trr-62.de/>
- [Casas:2010 2011] : *CASAS Smart Home*. 2011. – URL <http://ailab.wsu.edu/casas/>
- [COMPANIONSPROJEKT 2011] : *Companions Project*. 2011. – URL <http://www.companions-project.org/>
- [WILKS 2011] : *Yorick Wilks: Social Companions / Companion Technology*. 2011. – URL <http://staffwww.dcs.shef.ac.uk/people/Y.Wilks/>
- [Abowd u. a. 1999] ABOWD, Gregory ; DEY, Anind ; BROWN, Peter ; DAVIES, Nigel ; SMITH, Mark ; STEGGLES, Pete: Towards a Better Understanding of Context and Context-Awareness. In: GELLERSEN, Hans-W. (Hrsg.): *Handheld and Ubiquitous Computing* Bd. 1707. Springer Berlin / Heidelberg, 1999, S. 304–307. – URL [http://dx.doi.org/10.1007/3-540-48157-5\\_29](http://dx.doi.org/10.1007/3-540-48157-5_29)
- [Allen 1983] ALLEN, James F.: Maintaining knowledge about temporal intervals. In: *Commun. ACM* 26 (1983), Nr. 11, S. 832–843. – ISSN 0001-0782
- [Bedrouni u. a. 2009] BEDROUNI, Abdellah ; MITTU, Ranjeev ; BOUKHTOUTA, A. ; BERGER, Jean: *Distributed Intelligent Systems A Coordination Perspective*. Springer, 2009
- [Bohlken u. a. 2011] BOHLKEN, Wilfried ; KOOPMANN, Patrick ; NEUMANN, Bernd: SCENIOR: Ontology-based Interpretation of Aircraft Service Activities / Universität Hamburg. 2011 (Report FBI-HH-B-297 11). – Technical Report
- [Ellenberg 2010] ELLENBERG, Jens: Ein Wecker in einem ubicom Haus / HAW Hamburg. 2010. – Seminararbeit (AW1)
- [Ellenberg 2011a] ELLENBERG, Jens: Klassifizierung von Kontext in einer intelligenten Wohnung / HAW Hamburg. URL <http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master10-11-seminar/ellenberg/bericht.pdf>, 2011. – Forschungsbericht
- [Ellenberg 2011b] ELLENBERG, Jens: *Ontologiebasierte Aktivitätserkennung im Smart Home Kontext*. 2011. – Masterthesis, Abgabe vorraussichtlich 12/2011

- [Ellenberg u. a. 2011] ELLENBERG, Jens ; KARSTAEDT, Bastian ; VOSKUH, Sören ; LUCK, Kai von ; WENDHOLT, Birgit: *An Environment for Context-Aware Applications in Smart Homes*. 2011. – Vorgesehen für die „International conference on indoor positioning and indoor navigation“ am 21.-23.Sep 2011
- [Erman u. a. 1980] ERMAN, Lee D. ; HAYES-ROTH, Frederick ; LESSER, Victor R. ; REDDY, D. R.: The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. In: *ACM Comput. Surv.* 12 (1980), June, S. 213–253. – URL <http://doi.acm.org/10.1145/356810.356816>. – ISSN 0360-0300
- [Gottmann und Nachtigall 2011] GOTTMANN, Susann ; NACHTIGALL, Nico: *Modelling the Living Place Project using Algebraic Higher Order Nets*, HAW Hamburg, Diplomarbeit, June 2011
- [Greenberg 2001] GREENBERG, Saul: Context as a dynamic construct. In: *Hum.-Comput. Interact.* 16 (2001), December, S. 257–268. – URL [http://dx.doi.org/10.1207/S15327051HCI16234\\_09](http://dx.doi.org/10.1207/S15327051HCI16234_09). – ISSN 0737-0024
- [Ha u. a. 2007] HA, Young-Guk ; SOHN, Joo-Chan ; CHO, Young-Jo: ubiHome: An Infrastructure for Ubiquitous Home Network Services. Juni 2007. – Forschungsbericht. – 6 S
- [Janse 2008] JANSE, Dr. Maddy D.: *Amigo Projektseite*. 2008. – URL <http://www.hitech-projects.com/euprojects/amigo/index.htm>. – Philips Research
- [Karstaedt 2011a] KARSTAEDT, Bastian: Entwicklung eines Indoor Spatial Information Services für Smart Homes auf Basis der Industry Foundation Classes / HAW Hamburg. 2011. – Forschungsbericht
- [Karstaedt 2011b] KARSTAEDT, Bastian: Projektbericht 2: Entwicklung und Integration des Indoor Spatial Information Services in den Living Place Hamburg / HAW Hamburg. 2011. – Forschungsbericht
- [Krokhin u. a. 2003] KROKHIN, Andrei ; JEAVONS, Peter ; JONSSON, Peter: Reasoning about temporal relations: The tractable subalgebras of Allen’s interval algebra. In: *J. ACM* 50 (2003), Nr. 5, S. 591–640. – ISSN 0004-5411
- [von Luck u. a. 2010] LUCK, Prof. Dr. K. von ; KLEMKE, Prof. Dr. G. ; GREGOR, Sebastian ; RAHIMI, Mohammad A. ; VOGT, Matthias: *Living Place Hamburg – A place for concepts of IT based modern living* / Hamburg University of Applied Sciences. URL <http://livingplace.>

- [informatik.haw-hamburg.de/content/LivingPlaceHamburg\\_en.pdf](http://informatik.haw-hamburg.de/content/LivingPlaceHamburg_en.pdf), Mai 2010. – Forschungsbericht
- [Martin u. a. 2006] MARTIN, David ; BURSTEIN, Mark ; DENKER, Grit ; ELENUS, Daniel ; GIAMPAPA, Joseph ; MCDERMOTT, Drew ; MCGUINNESS, Deborah ; MCILRAITH, Sheila ; PAOLUCCI, Massimo ; PARSIA, Bijan ; PAYNE, Terry ; SIRIN, Evren ; SRINIVASAN, Naveen ; SYCARA, Katia: *OWL-S 1.2 Release*. 2006. – URL <http://www.ai.sri.com/daml/services/owl-s/>
- [Moran und Dourish 2001] MORAN, Thomas P. ; DOURISH, Paul: Introduction to this special issue on context-aware computing. In: *Hum.-Comput. Interact.* 16 (2001), December, S. 87–95. – URL [http://dx.doi.org/10.1207/S15327051HCI16234\\_01](http://dx.doi.org/10.1207/S15327051HCI16234_01). – ISSN 0737-0024
- [Nakajima und Satoh 2006] NAKAJIMA, Tatsuo ; SATOH, Ichiro: A software infrastructure for supporting spontaneous and personalized interaction in home computing environments. In: *Personal Ubiquitous Comput.* 10 (2006), Nr. 6, S. 379–391. – ISSN 1617-4909
- [Nau u. a. 2004] NAU, Dana ; GHALLAB, Malik ; TRAVERSO, Paolo: *Automated Planning: Theory & Practice*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2004. – ISBN 1558608567
- [Nebel und Bürckert 1995] NEBEL, Bernhard ; BÜRCKERT, Hans-Jürgen: Reasoning about temporal relations: a maximal tractable subclass of Allen’s interval algebra. In: *J. ACM* 42 (1995), Nr. 1, S. 43–66. – ISSN 0004-5411
- [Neumann und Terzic 2011] NEUMANN, Bernd ; TERZIC, Kasim: Context-based Probabilistic Scene Interpretation / Universität Hamburg. 09 2011. – Technical Report
- [Odersky u. a. 2008] ODERSKY, Martin ; SPOON, Lex ; VENNERS, Bill: *Programming in Scala*. artima press, 2008
- [Otto und Voskuhl 2010] OTTO, Kjiell ; VOSKUHL, Sören: Entwicklung einer Architektur für das Livingplace Hamburg / HAW Hamburg. 2010. – Forschungsbericht
- [Padberg 1992 – 2010] PADBERG, Julia: *Publikationen Julia Padberg (TU Trier)*. 1992 - 2010. – URL <http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/p/Padberg:Julia.html>
- [Schank und Abelson 1977] SCHANK, Roger C. ; ABELSON, Robert P.: *Scripts, plans, goals and understanding: an inquiry into human knowledge structures*. John Wiley & Sons Inc, 1977

- [Selker 2007] SELKER, Ted: *Context Aware Computing: Understanding and Responding to Human Interactions (Human Computer Interaction Seminar, Fall 2007)*. iTunes U. November 2007. – URL <http://itunes.apple.com/de/itunes-u/context-aware-computing-understanding/id384230159?i=85092068>. – Stanford University
- [Tennstedt 2007] TENNSTEDT, Sven: *Genese sozialen Verhaltens in Multiagentensystemen durch genetische Verfahren*. 2007. – URL <http://users.informatik.haw-hamburg.de/~kvl/tennstedt/bachelor.pdf>
- [Vallée 2009] VALLÉE, M.: *UN INTERGICIEL MULTI-AGENT POUR LA COMPOSITION FLEXIBLE D'APPLICATIONS EN INTELLIGENCE AMBIANTE*, Ecole Nationale Supérieure des Mines, Dissertation, Januar 2009
- [Vallée u. a. 2005a] VALLÉE, M. ; RAMPARANY, F. ; VERCOUTER, L.: Dynamic service composition in ambient intelligence environments: a multi-agent approach. In: *Proceeding of the First European Young Researcher Workshop on Service-Oriented Computing* (2005), April
- [Vallée u. a. 2005b] VALLÉE, M. ; RAMPARANY, F. ; VERCOUTER, L.: A multi-agent system for dynamic service composition in ambient intelligence environments. In: *Advances in Pervasive Computing, Adjunct Proceedings of the Third International Conference on Pervasive Computing (Pervasive 2005)* (2005), Mai
- [Voskuhl 2010] VOSKUHLE, Sören: *Bereitstellung einer Sensorwolke / HAW Hamburg*. 2010. – Forschungsbericht
- [de Weerd t u. a. 2005] WEERDT, Mathijs de ; MORS, Adriaan ter ; WITTEVEEN, Cees: Multi-agent Planning: An introduction to planning and coordination. In: *Handouts of the European Agent Summer School*, URL <http://www.st.ewi.tudelft.nl/~mathijs/publications/easss05.pdf>, 2005, S. 1–32
- [Weiser 1991] WEISER, Mark: *The Computer for the 21st Century / XEROX*. September 1991. – Forschungsbericht
- [Winograd 2001] WINOGRAD, Terry: Architectures for context. In: *Hum.-Comput. Interact.* 16 (2001), December, S. 401–419. – URL [http://dx.doi.org/10.1207/S15327051HCI16234\\_18](http://dx.doi.org/10.1207/S15327051HCI16234_18). – ISSN 0737-0024



## A. Inhalt der CD

Dieser Arbeit ist eine CD beigelegt. Diese enthält:

- die vorliegende Masterarbeit als PDF,
- den Quellcode des Prototypen, inklusive des LivingPlaceAdapters für die ActiveMQ,
- viele von den referenzierten Dokumenten aus dem Literaturverzeichnis und darüber hinaus weitere interessante Paper.

## B. Scala Schnellübersicht

Im Folgenden sollen grundlegende Regeln der Scala-Syntax vermittelt werden, um die Codebeispiele in dieser Arbeit verstehen zu können. Die folgenden Erklärungen der Scala-Syntax setzen voraus, dass das Lesen von Java Code grundsätzlich keine Schwierigkeiten bereitet, da oft auf die Unterschiede zur Java-Syntax hingewiesen wird. Weiterführende Informationen sind [Odersky u. a. \(2008\)](#) oder der Website zu Scala (<http://www.scala-lang.org>) zu entnehmen.

### B.1. Deklaration von Klassen

Die Deklaration von Klassen ist ähnlich wie in Java. Größter Unterschied ist, dass auf einen Constructor verzichtet wird und Parameter zur Erstellung einer Klasse gleich hinter dem Klassennamen geschrieben werden können.

Eine Klassendeklaration in Java:

```
1 public class SomeClass {
2     String foo;
3     Integer bar;
4     public SomeClass(String foo, Integer bar) {
5         this.foo = foo;
6         this.bar = bar;
7     }
8 }
```

Die Äquivalenz in Scala ist folgende:

```
1 class SomeClass(foo : String , bar : Int) {
2 }
```

In Scala gibt es zusätzlich sogenannte Case-Classes, die bei der Deklaration mit dem Schlüsselwort `case` gekennzeichnet werden. Das Besondere an Case-Classes ist u. a., dass neue Instanzen einer Case-Class mit den selben Parametern für die Konstruktion auch gleich sind. Folgendes ergibt also `true`:

```
1 case class Goal(goal_id : String , resource : String)
2
3 new Goal("g1", "heat") == new Goal("g1", "heat") // ergibt true
```

## B.2. Singletons

In Scala können mit dem Keyword `object` Singletons erstellt werden. Die Deklaration eines Singleton erfolgt dann analog zu dem einer Klasse. Mit der Deklaration sind Singletons im System verwendbar. Sie werden über ihren Namen aufgerufen.

Ein Beispiel:

```
1 object SomeSingleton {
2   def do(foo: String): String = {
3     foo + foo
4   }
5 }
6
7 SomeSingleton.do("Hello_World!") // returns "Hello World!Hello World!"
```

## B.3. Generics

Anders als in Java werden Generics in Scala nicht mit spitzen Klammern, sondern mit eckigen gekennzeichnet.

Die beiden folgenden Ausdrücke sind äquivalent:

Java:

```
1 List<Integer> someList;
```

Scala:

```
1 List[Int] someList;
```

## B.4. var/val: Variablen und Werte

In Scala wird zwischen Values (Keyword `val`) und Variablen (Keyword `var`) unterschieden. Values sind vergleichbar mit der Java Deklaration „final Type bezeichner“, sie sind nur einmal zuweisbar und behalten ihren Wert bis der Scope verlassen wird. Variablen verhalten sich so wie von Java bekannt.

## B.5. Functions

Aufbau:

```
1 def foo (parameter : ParameterType) : ReturnType = {  
2   var bar  
3   // place code here  
4   bar  
5 }
```

Letzter Wert/letzte Variable werden in Functions auch ohne das Keyword `return` zurück gegeben. Dennoch kann das Keyword `return` verwendet werden.

## B.6. Partial Functions

Partial Functions werden zum Beispiel als Parameter bei den Funktionen `foreach` oder `collect` übergeben.

Aufbau:

```
1 { parameter => body }
```

Die Funktion `foreach` verhält sich im Prinzip so wie in Java der Ausdruck `for (Type laufvariable: someCollection) {}`. In Scala wird dieses Verhalten nur mit einer Partial Function als Parameter für die Funktion `foreach` realisiert, da in Scala kein Keyword `for` existiert.

Beispiel `foreach`:

```
1 someCollection.foreach ({ foo => println (foo) })
```

Hier ist zu beachten, dass die Übergabe von Parametern an Funktionen nicht zwingend in runden Klammern stehen müssen. Der folgende Ausdruck ist also ebenso korrekt:

```
1 someCollection.foreach { foo => println (foo) }
```

Mit `collect` kann aus einer gegebenen Collection eine neue erstellt werden, indem auf jedes Element eine Funktion angewandt wird, die ein neues Element zurück gibt. Zusätzlich können mit dem Keyword `case` oder der Keyword-Kombination `case ...if` verschiedene Fälle abgedeckt werden.

Beispiel `collect` mit `case`:

```
1 someCollection.collect {  
2   case foo: Int => foo  
3   case foo: String => foo.toInt  
4 }
```

Das Gleiche geht auch mit einer Map (zurück gegeben wird eine `List [Int]`):

```
1 someMap.collect {
2   case (key: String, value: Int) => value
3   case (key: String, value: String) => value.toInt
4 }
```

Über das Keyword `if` in einer `case ...if` Kombination, kann ein `case` aktiviert oder deaktiviert werden.

Beispiel `collect` mit `case ...if` (Typbezeichnungen können weggelassen werden, wenn der Typ klar ist):

```
1 someCollection.collect {
2   case foo if foo < 5 => foo * 2
3   case foo if foo > 5 => foo * 3
4 }
```

Zusätzlich können noch Jokerzeichen eingesetzt werden. Angenommen es existiert eine Liste von Tupeln, in Scala als `List[Tuple2[Int, Int]]` (Beispielhaft besteht das Tupel aus zwei Ganzzahlen) definiert. Soll nun über alle Tupel iteriert werden und abhängig von den Werten in den Elementen der Tupel entweder das erste oder zweite Element des jeweiligen Tupels zurückgegeben werden, so kann das folgendermaßen geschehen:

```
1 someTupleList.collect {
2   case (first, _) if first > 5 => first
3   case (_, second) if second < 5 => second
4 }
```

Es wird eine Liste von Ganzzahlen zurückgegeben. Der Unterstrich `»_«` dient als Jokerzeichen, der dazu führt, dass das betreffende Element nicht beachtet wird.

## C. Quellcode

### C.1. Goal

```
1 package org.livingplace.workflow
2
3 import scala.collection.mutable.Map
4
5 /**
6  * @author Sven Tennstedt
7  *
8  */
9 case class Goal(gid:String , ro:String , rs:String , m:Boolean , t:GoalType) {
10   val goal_id: String = gid
11   val resource: String = ro
12   val resource_state: String = rs
13   val mandatory: Boolean = m
14   val goal_type: GoalType = t
15   var time_start: String = ""
16   var time_end: String = ""
17   private[this] var _dependency: Set[String] = Set[String]()
18
19   def dependency: String = _dependency.size match {
20     case 0 => ""
21     case _ => _dependency.addString(new StringBuilder(), ",").toString
22   }
23
24   def dependency_(value: String) {
25     value.length match {
26       case 0 => _dependency = Set.empty[String]
27       case _ => value.split(",").foreach{ gid => _dependency += gid }
28     }
29   }
30
31   def dependencySet(): Set[String] = _dependency.toSet
32
33   def addDependency(from: Goal) = {
```

```
34     _dependency += from.goal_id
35   }
36
37   def hasTimeConstraints: Boolean = {
38     time_start.length() + time_end.length() > 0
39   }
40
41   override def toString(): String = {
42     gid + ":_␣" + resource + "(" + resource_state + ")_␣" + goal_type
43   }
44 }
45
46 object Goal {
47   def fromMap(values: scala.collection.immutable.Map[String, Any]): Goal = {
48     val goal_id: String = values("goal_id").asInstanceOf[String]
49     val resource: String = values("resource").asInstanceOf[String]
50     val resource_state: String = values("resource_state").asInstanceOf[String]
51     val mandatory: Boolean = values("mandatory").asInstanceOf[Boolean]
52     val goal_type_str: String = values("goal_type").asInstanceOf[String]
53     val goal_type: GoalType = GoalTypes.valueOf(goal_type_str)
54
55     val goal = new Goal(goal_id,
56                       resource,
57                       resource_state,
58                       mandatory,
59                       goal_type)
60
61     val dependency: String = values("dependency").asInstanceOf[String]
62     val time_start: String = values("time_start").asInstanceOf[String]
63     val time_end: String = values("time_end").asInstanceOf[String]
64
65     goal.dependency = dependency
66     goal.time_start = time_start
67     goal.time_end = time_end
68     goal
69   }
70
71   def toMap(goal: Goal): scala.collection.immutable.Map[String, Any] = {
72     val values: Map[String, Any] = Map[String, Any]()
73     values.put("goal_id", goal.goal_id)
74     values.put("resource", goal.resource)
75     values.put("resource_state", goal.resource_state)
76     values.put("mandatory", goal.mandatory)
```

```
77 values.put("goal_type", goal.goal_type.name)
78 values.put("dependency", goal.dependency)
79 values.put("time_start", goal.time_start)
80 values.put("time_end", goal.time_end)
81 values.toMap[String, Any]
82 }
83 }
```

## C.2. GoalTypes

```
1 package org.livingplace.workflow
2
3 import org.livingplace.workflow.misc._
4
5 /**
6  * @author Sven Tennstedt
7  *
8  */
9 object GoalTypes extends Enumeration[GoalType] {
10   case object UNTIL_END extends GoalType("UNTIL_END")
11   add(UNTIL_END)
12   case object EXACTLY_TO_END extends GoalType("EXACTLY_TO_END")
13   add(EXACTLY_TO_END)
14   case object FROM_START extends GoalType("FROM_START")
15   add(FROM_START)
16 }
17
18 sealed case class GoalType(t: String) extends Enumeration.Value(t) {
19   val value: String = t
20 }
```

## C.3. GoalRelation

```
1 package org.livingplace.workflow.intervals
2
3 import org.livingplace.workflow._
4
5 /**
6  * @author Sven Tennstedt
7  *
8  */
9 case class GoalRelation(fromGoal: Goal, r: Set[Constraint.Value], toGoal: Goal)
10 {
```



```
11 toGoal.addDependency(fromGoal)
12
13 val from: String = fromGoal.goal_id
14 val to: String = toGoal.goal_id
15 val constraints: Set[Constraint.Value] = r
16
17 def constraints_intersect(constraints: Set[Constraint.Value]):
18     Set[Constraint.Value] = {
19     this.constraints.intersect(constraints)
20 }
21
22 override def toString = {
23     from + "└─{" + constraints.mkString + "}->└" + to
24 }
25 }
```

## C.4. Scenario

```
1 package org.livingplace.workflow
2
3 import org.livingplace.workflow.intervals._
4 import org.livingplace.workflow.intervals.Constraint._
5 import scala._
6
7 /**
8  *
9  * @author Sven Tennstedt
10  *
11  */
12 class Scenario(constraints: Set[GoalRelation]) {
13     val bindingOne: Set[Constraint.Value] = Set[Constraint.Value](b, m, o)
14
15     val bindingOnePlus = Set[Constraint.Value](o, m, di, fi)
16
17     val bindingInclusions = Set[Constraint.Value](fi, di)
18
19     val allenMap: Map[Tuple2[String, String], GoalRelation] =
20         mkAllenMap(constraints)
21     private [this]
22     def mkAllenMap(rs: Set[GoalRelation]):
23         Map[Tuple2[String, String], GoalRelation] = {
24         if (rs.isEmpty)
25             return Map[Tuple2[String, String], GoalRelation]()
```

### C. Quellcode

---

```
26     val relation = rs.first
27     return mkAllenMap(rs.tail) + ((relation.from, relation.to) -> relation)
28 }
29
30 val goalMap: Map[String, Goal] = mkGoalMap(constraints)
31 private[this] def mkGoalMap(rs: Set[GoalRelation]): Map[String, Goal] = {
32     if (rs.isEmpty)
33         return Map[String, Goal]()
34     val relation = rs.first
35     return mkGoalMap(rs.tail)
36         + (relation.from -> relation.fromGoal)
37         + (relation.to -> relation.toGoal)
38 }
39
40 /*
41  *
42  */
43
44 // realy solved
45 var markedGreen: Set[String] = Set()
46
47 // marked for solving
48 var markedYello: Set[String] = Set()
49
50 def markSolved(gid: String): Unit = {
51     doForGoal(gid, (g: String) => markedGreen += g)
52 }
53
54 def markForSolving(gid: String): Unit = {
55     doForGoal(gid, (g: String) => markedYello += g)
56 }
57
58 def markedSolved: Set[String] = markedGreen
59
60 def hasMarkedGoals(): Boolean = {
61     !markedGreen.isEmpty
62 }
63
64 def doForGoal(goalId: String, func: (String) => Unit): Unit = {
65     val goal = getGoal(goalId)
66     if (goal != null) {
67         func(goalId)
68     }
```

```

69 }
70
71 def getGoal(goalId: String): Goal = {
72   if (goalMap.contains(goalId))
73     return goalMap(goalId)
74   null
75 }
76
77 /**
78  * collect all goals that have no incoming edges
79  */
80 def findStartingGoals: Set[Goal] = {
81   val toGoals = constraints.collect { case x => x.to }
82   val rootGoals = constraints.collect {
83     case x: GoalRelation if !toGoals.contains(x.from) => x.fromGoal
84   }
85
86   var startingGoals = Set() ++ rootGoals
87
88   rootGoals.foreach {
89     goal =>
90       startingGoals += allenMap.collect {
91         case ((_, _), gr) if isBindable(goal, gr.toGoal) => gr.toGoal
92       }
93   }
94
95   startingGoals
96 }
97
98 def getDirectReachableGoalsFrom(goal_id: String): Set[String] = {
99   Set[String]() ++ allenMap.collect {
100     case ((from, to), _)
101       if from == goal_id
102       => to
103   }
104 }
105
106 def getRelationsForDirectReachableGoalsFrom(goal: Goal):
107   Set[GoalRelation] =
108   getRelationsForDirectReachableGoalsFrom(goal.goal_id)
109 def getRelationsForDirectReachableGoalsFrom(goal_id: String):
110   Set[GoalRelation] = {
111   Set[GoalRelation]() ++ allenMap.collect {

```

```

112     case ((from, _), gr)
113         if from == goal_id
114             => gr
115     }
116 }
117
118 def getReachableRelationsFrom(goal: Goal): Set[GoalRelation] = {
119     Set[GoalRelation]() ++ allenMap.collect({
120         case ((from, _), gr)
121             if from == goal.goal_id &&
122                 gr.constraints_intersect(bindingOne).nonEmpty
123             => gr
124         case ((from, _), gr)
125             if from != goal.goal_id && isBindable(goal, gr.toGoal)
126             => reachableRelation(goal, gr.toGoal)
127     })
128 }
129
130 def isNotMarked(goalId: String): Boolean = !isMarked(goalId)
131 def isMarked(goalId: String): Boolean = markedGreen.contains(goalId)
132
133 def isBindable(goal1: Goal, goal2: Goal): Boolean = {
134     reachableRelation(goal1, goal2).constraints.nonEmpty
135 }
136
137 def reachableRelation(goal1: Goal, goal2: Goal): GoalRelation = {
138     val shortest_path = shortestPath(goal1, goal2, List())
139     if (shortest_path.isEmpty)
140         return GoalRelation(goal1, Set(), goal2)
141
142     val fromFirst = transitivity(goal1, goal2)
143     val fromFirstBinding = fromFirst.constraints_intersect(bindingOnePlus)
144     if (fromFirstBinding.nonEmpty)
145         return fromFirst
146
147     val startingRelation = shortest_path.first
148     val firstBindsStrong =
149         startingRelation.constraints_intersect(bindingInclusions)
150     if (firstBindsStrong.nonEmpty) {
151         val fromSecond = transitivity(startingRelation.toGoal, goal2)
152         val fromSecondBinding =
153             fromSecond.constraints_intersect(bindingOnePlus)
154         if (fromSecondBinding.nonEmpty)

```

```

155     return fromSecond
156   } else {
157     val fromSecond = transitivity(startingRelation.toGoal, goal2)
158     val fromSecondBinding =
159       fromSecond.constraints_intersect(bindingInclusions)
160     if (fromSecondBinding.nonEmpty)
161       return fromSecond
162   }
163
164   GoalRelation(goal1, Set(), goal2)
165 }
166
167 def transitivity(goal1: Goal, goal2: Goal): GoalRelation = {
168   val shortest_path = shortestPath(goal1, goal2, List())
169
170   if (shortest_path.isEmpty)
171     return GoalRelation(goal1, Set(), goal2)
172
173   val firstConstraints = shortest_path.first.constraints
174   val tail = shortest_path.tail
175   var transitivity = firstConstraints
176   tail.foreach { gr =>
177     transitivity = AllenTransitivity.of(transitivity, gr.constraints)
178   }
179   GoalRelation(goal1, transitivity, goal2)
180 }
181
182 def shortestPath(goal1: Goal, goal2: Goal, path: List[GoalRelation]):
183   List[GoalRelation] = {
184   val key = (goal1.goal_id, goal2.goal_id)
185   if (allenMap.contains((goal1.goal_id, goal2.goal_id))) {
186     val result_path = path :+ allenMap(key)
187     return result_path
188   }
189
190   val follower = allenMap.collect {
191     case ((from, _), gr)
192       if from == goal1.goal_id
193       => gr.toGoal
194   }
195   follower.foreach { f =>
196     val result_path = path :+ allenMap((goal1.goal_id, f.goal_id))
197     val shortest_path = shortestPath(f, goal2, result_path)

```

```

198     if (shortest_path.size > 0) {
199         return shortest_path
200     }
201 }
202
203 List[GoalRelation]()
204 }
205
206 def paths(goal1: Goal, goal2: Goal, path: List[GoalRelation]):
207     Set[List[GoalRelation]] = {
208     val key = (goal1.goal_id, goal2.goal_id)
209     var result_paths = Set[List[GoalRelation]()
210     val follower = allenMap.collect {
211         case ((from, _), gr)
212             if from == goal1.goal_id
213             => gr.toGoal
214     }
215     follower.foreach { f =>
216         val result_path = path :+ allenMap((goal1.goal_id, f.goal_id))
217         result_paths = result_paths ++ paths(f, goal2, result_path)
218     }
219
220     if (allenMap.contains((goal1.goal_id, goal2.goal_id))) {
221         val result_path = path :+ allenMap(key)
222         return result_paths + result_path
223     }
224
225     return result_paths
226 }
227 }

```

## C.5. GoalManagerGraph

```

1 package org.livingplace.workflow.reasoning
2
3 import org.livingplace.workflow.intervals._
4 import org.livingplace.workflow.intervals.Constraint._
5 import org.livingplace.workflow._
6
7 /**
8  * @author Sven Tennstedt
9  *
10 */

```

```
11 class GoalManagerGraph(_scenario: Scenario) {
12   val scenario: Scenario = _scenario
13
14   def markSolved(goalId: String) = {
15     scenario.markSolved(goalId)
16   }
17
18   def getNextGoalsToBeReached(): Set[Goal] = {
19     if (!scenario.hasMarkedGoals()) {
20       return scenario.findStartingGoals
21     }
22
23     var reachable = Set[String]()
24     var marked: Set[String] = Set[String]()
25     marked += scenario.markedSolved
26     marked.foreach { gid =>
27       val next = resolve(gid)
28       reachable += next
29     }
30
31     reachable.collect {
32       case gid if scenario.isNotMarked(gid) => scenario.goalMap(gid)
33     }
34   }
35
36   def resolve(goal_id: String): Set[String] = {
37     var result: Set[String] = Set[String]()
38     val goal: Goal = scenario.goalMap(goal_id)
39     var relations = scenario.getReachableRelationsFrom(goal)
40     relations.foreach { r: GoalRelation =>
41       result += r.to
42     }
43     result
44   }
45 }
```

## D. Tests und Versuchsaufbau

### D.1. Unittests für den GoalManager

In diesem Abschnitt werden die Unittests gezeigt, mit denen überprüft wurde, ob der GoalManager die richtigen Ziele auswählt.

#### D.1.1. Unittests Userstory 1

```
1 class GoalManagerUserStory1Test {
2   val g1 = new Goal("g1", "heat", "at_daymode", true, EXACTLY_TO_END)
3   val g2 = new Goal("g2", "human", "is_aware", true, EXACTLY_TO_END)
4   val g3 = new Goal("g3", "human", "is_showered", true, EXACTLY_TO_END)
5   val g4 = new Goal("g4", "human", "is_dressed", true, UNTIL_END)
6   val g5 = new Goal("g5", "coffee", "is_cooked", true, UNTIL_END)
7   val g6 = new Goal("g6", "human", "have_had_breakfast", true, UNTIL_END)
8   val g7 = new Goal("g7", "human",
9     "is_informed_about_day's_schedule", true, UNTIL_END)
10  val g8 = new Goal("g8", "documents", "are_provided", true, FROM_START)
11
12  val gr1 = GoalRelation(g1, Set(b, o, fi), g2).withAddingDependency
13  val gr2 = GoalRelation(g2, Set(b, m), g3).withAddingDependency
14  val gr3 = GoalRelation(g3, Set(b, m), g4).withAddingDependency
15  val gr4 = GoalRelation(g4, Set(b, m, o, fi), g5).withAddingDependency
16  val gr5 = GoalRelation(g5, Set(b, m), g6).withAddingDependency
17  val gr6 = GoalRelation(g6, Set(b, m, o), g7).withAddingDependency
18  val gr7 = GoalRelation(g4, Set(b, m), g8).withAddingDependency
19  val gr = Set[GoalRelation](gr1, gr2, gr3, gr4, gr5, gr6, gr7)
20
21  @Test
22  def testFindMarkableGoals1 = {
23    val gm = new GoalManagerGraph(new Scenario(gr))
24
25    val goals: Set[Goal] = gm.getNextGoalsToBeReached
26    assertEquals(Set(g1, g2, g3), goals)
27  }
```



```
28
29 @Test
30 def testFindMarkableGoals2 = {
31     val gm = new GoalManagerGraph(new Scenario(gr))
32     gm.markSolved(g1.goal_id)
33
34     val goals: Set[Goal] = gm.getNextGoalsToBeReached
35     assertEquals(Set(g2, g3), goals)
36 }
37
38 @Test
39 def testFindMarkableGoals3 = {
40     val gm = new GoalManagerGraph(new Scenario(gr))
41     gm.markSolved(g2.goal_id)
42
43     val goals: Set[Goal] = gm.getNextGoalsToBeReached
44     assertEquals(Set(g3), goals)
45 }
46
47 @Test
48 def testFindMarkableGoals4 = {
49     val gm = new GoalManagerGraph(new Scenario(gr))
50     gm.markSolved(g3.goal_id)
51
52     val goals: Set[Goal] = gm.getNextGoalsToBeReached
53     assertEquals(Set(g4, g5), goals)
54 }
55
56 @Test
57 def testFindMarkableGoals5 = {
58     val gm = new GoalManagerGraph(new Scenario(gr))
59     gm.markSolved(g4.goal_id)
60
61     val goals: Set[Goal] = gm.getNextGoalsToBeReached
62     assertEquals(Set(g5, g6, g8), goals)
63 }
64
65 @Test
66 def testFindMarkableGoals6 = {
67     val gm = new GoalManagerGraph(new Scenario(gr))
68     gm.markSolved(g5.goal_id)
69     gm.markSolved(g6.goal_id)
70
```

```
71     val goals: Set[Goal] = gm.getNextGoalsToBeReached
72     assertEquals(Set(g7), goals)
73 }
74
75 @Test
76 def testFindMarkableGoals7 = {
77     val gm = new GoalManagerGraph(new Scenario(gr))
78     gm.markSolved(g7.goal_id)
79     gm.markSolved(g8.goal_id)
80
81     val goals: Set[Goal] = gm.getNextGoalsToBeReached
82     assertEquals(Set(), goals)
83 }
84 }
```

### D.1.2. Unittests Userstory 2

```
1 class GoalManagerUserStory2Test {
2     val g1 = new Goal("g1", "calendar",
3         "visitors_appointment_existend", true, FROM_START)
4     val g1a = new Goal("g1a", "calendar",
5         "appointment_type_assigned", true, UNTIL_END)
6     val g2 = new Goal("g2", "human", "has_choosed_dish", true, UNTIL_END)
7     val g2a = new Goal("g2a", "homemaker",
8         "list_of_meals_composed", true, FROM_START)
9     val g3 = new Goal("g3", "homemaker",
10        "stock_of_food_assured", true, EXACTLY_TO_END)
11    val g3a = new Goal("g3a", "homemaker", "shopping_list", true, FROM_START)
12    val g3b = new Goal("g3b", "human", "shopping_done", true, UNTIL_END)
13
14    val gr1 = GoalRelation(g1, Set(di), g1a).withAddingDependency
15    val gr2 = GoalRelation(g1a, Set(b, m), g2).withAddingDependency
16    val gr3 = GoalRelation(g2, Set(di), g2a).withAddingDependency
17    val gr4 = GoalRelation(g2, Set(b, m), g3).withAddingDependency
18    val gr5 = GoalRelation(g3, Set(di, fi), g3a).withAddingDependency
19    val gr6 = GoalRelation(g3, Set(di, fi), g3b).withAddingDependency
20    val gr7 = GoalRelation(g3a, Set(fi), g3b).withAddingDependency
21    val gr = Set[GoalRelation](gr1, gr2, gr3, gr4, gr5, gr6, gr7)
22
23    @Test
24    def testFindMarkableGoals1 = {
25        val gm = new GoalManagerGraph(new Scenario(gr))
26
27        val goals: Set[Goal] = gm.getNextGoalsToBeReached
```

```
28   assertEquals(Set(g1, g1a, g2, g2a, g3, g3a, g3b), goals)
29   }
30 }
```

## D.2. GoalAgent Beispiel

```
1 class GoalAgent1(topic: String, scenarioTopic: String)
2   extends AbstractGoalAgentTopicActor(topic, scenarioTopic) {
3   override def handleMessage(msg: Message[GoalMessageType]) = {
4     msg match {
5       case Message(GOAL_PLANNED) if handles == None => {
6         val goal: Goal = Goal.fromMap(msg.data)
7         if (goal.resource == "human" && goal.resource_state == "awake") {
8           handles = goal
9           val logMsg = "consume_goal:_ " + goal
10          printMessage(logMsg)
11          addPendingDependency(goal)
12          printMessage("added_dependencies_" + dependencyGoalIds)
13        }
14      }
15      case Message(GOAL_TIMEBOUNDARIES) if handles != None => {
16        val goal: Goal = Goal.fromMap(msg.data)
17        if (isPendingDependency(goal)) {
18          val logMsg = "consume_goal_timeboundaries_from_" + goal
19          printMessage(logMsg)
20          update_timeboundaries(goal)
21        }
22      }
23      case Message(GOAL_SOLVED) if handles != None => {
24        val goal: Goal = Goal.fromMap(msg.data)
25        if (handles == goal) {
26          handles = None
27        }
28      }
29      case _ =>
30    }
31  }
32
33  override def handleGoal: Unit = {
34    printMessage("handle_goal_" + handles);
35
36    handles.time_start = "07.45"
37    handles.time_end = "08.00"
```

```
38
39     val msg = Message(GOAL_TIMEBOUNDARIES);
40     msg.data = Goal.toMap(handles)
41     send(msg)
42 }
43 }
```

### D.3. Console-Log

Das Console-Log zeigt die Ausgabe, die die Agenten während ihrer Laufzeit generieren.

<b>consume</b>	Agent empfängt und verarbeitet eine Nachricht. Der Nachrichtentyp steht direkt dahinter.
<b>timeboundaries calculated</b>	Der Agent kennt alle Zeiten der Ziele, von denen sein Ziel abhängt. Er konnte berechnen, in welchem Zeitrahmen sein Ziel erfüllt werden soll.
<b>added dependencies</b>	Der Agent merkt sich von welchen Zielen sein Ziel abhängt.
<b>handle</b>	Agent wird sein Ziel erfüllen.

#### D.3.1. : Versuch 1

Im ersten Versuch werden die Ziele in der Reihenfolge erfüllt, wie im GoalManager-Test Seite [67](#) beschrieben.

Konsolidiertes Console-Log:

```
1 18:49:32.240 [GoalManagerTopicAgent] scenario selected received!
2 18:49:32.293 [GoalManagerTopicAgent] sending GOAL_PLANNED : g1: heat(daymode) EXACTLY_TO_END
3 18:49:32.518 [GoalManagerTopicAgent] sending GOAL_PLANNED : g2: human(awake) EXACTLY_TO_END
4 18:49:32.523 [GoalManagerTopicAgent] sending GOAL_PLANNED : g3: human(has_had_a_shower) EXACTLY_TO_END
5 18:49:33.074 [GoalAgent2] consume GOAL_PLANNED : g1: heat(daymode) EXACTLY_TO_END
6 18:49:33.082 [GoalAgent2] added dependencies Map()
7 18:49:33.536 [GoalAgent1] consume GOAL_PLANNED : g2: human(awake) EXACTLY_TO_END
8 18:49:33.614 [GoalAgent1] added dependencies Map(g1 -> None: --)
9 18:49:33.646 [GoalAgent1b] consume GOAL_PLANNED : g3: human(has_had_a_shower) EXACTLY_TO_END
10 18:49:33.669 [GoalAgent1b] added dependencies Map(g2 -> None: --)
11 18:49:34.550 [GoalAgent2] timeboundaries calculated 07.00 to 08.00
12 18:49:35.615 [GoalAgent2] handle goal g1: heat(daymode) EXACTLY_TO_END 07.00 to 08.00
13 18:49:35.658 [GoalAgent1] consume GOAL_TIMEBOUNDARIES from g1: heat(daymode) EXACTLY_TO_END 07.00 to 08.00
14 18:49:35.658 [GoalAgent1] dependencies updated Map(g1 -> g1: heat(daymode) EXACTLY_TO_END 07.00 to 08.00)
15 18:49:37.037 [GoalAgent1] timeboundaries calculated 07.45 to 08.00
16 18:49:38.983 [GoalAgent1] handle goal g2: human(awake) EXACTLY_TO_END 07.45 to 08.00
17 18:49:39.061 [GoalAgent1b] consume GOAL_TIMEBOUNDARIES from g2: human(awake) EXACTLY_TO_END 07.45 to 08.00
18 18:49:39.062 [GoalAgent1b] dependencies updated Map(g2 -> g2: human(awake) EXACTLY_TO_END 07.45 to 08.00)
19 18:49:40.109 [GoalAgent1b] timeboundaries calculated 08.00 to 08.15
20 18:49:41.212 [GoalAgent1b] handle goal g3: human(has_had_a_shower) EXACTLY_TO_END 08.00 to 08.15
21 18:49:51.171 [GoalManagerTopicAgent] scenario goal solved received! goalId: g1
22 18:49:51.231 [GoalAgent2] consume GOAL_SOLVED and end handling for goal g1: heat(daymode) EXACTLY_TO_END
23 18:49:56.103 [GoalManagerTopicAgent] scenario goal solved received! goalId: g2
24 18:49:56.150 [GoalAgent1] consume GOAL_SOLVED and end handling for goal g2: human(awake) EXACTLY_TO_END
25 18:50:00.589 [GoalManagerTopicAgent] scenario goal solved received! goalId: g3
```

## D. Tests und Versuchsaufbau

```
26 18:50:00.634 [GoalManagerTopicAgent] sending GOAL_PLANNED : g4: human(is_dressed) UNTIL_END
27 18:50:00.645 [GoalManagerTopicAgent] sending GOAL_PLANNED : g5: coffee(cooked) UNTIL_END
28 18:50:00.655 [GoalAgent3] consume GOAL_PLANNED : g5: coffee(cooked) UNTIL_END
29 18:50:00.663 [GoalAgent3] added dependencies Map(g4 -> None: --)
30 18:50:00.677 [GoalAgent1b] consume GOAL_SOLVED and end handling for goal
                               g3: human(has_had_a_shower) EXACTLY_TO_END
31
32 18:50:00.701 [GoalAgent1c] consume GOAL_PLANNED : g4: human(is_dressed) UNTIL_END
33 18:50:00.710 [GoalAgent1c] added dependencies Map(g3 -> None: --)
34 18:50:00.734 [GoalAgent1c] consume GOAL_TIMEBOUNDARIES from g3: human(has_had_a_shower)
                               EXACTLY_TO_END 08.00 to 08.15
35
36 18:50:00.737 [GoalAgent1c] dependencies updated Map(g3 -> g3: human(has_had_a_shower)
                               EXACTLY_TO_END 08.00 to 08.15)
37
38 18:50:02.665 [GoalAgent1c] timeboundaries calculated 08.15 to 08.25
39 18:50:04.005 [GoalAgent1c] handle goal g4: human(is_dressed) UNTIL_END 08.15 to 08.25
40 18:50:04.037 [GoalAgent3] consume GOAL_TIMEBOUNDARIES from g4: human(is_dressed) UNTIL_END 08.15 to 08.25
41 18:50:04.037 [GoalAgent3] dependencies updated Map(g4 -> g4: human(is_dressed) UNTIL_END 08.15 to 08.25)
42 18:50:05.708 [GoalAgent3] timeboundaries calculated 08.15 to 08.25
43 18:50:06.850 [GoalAgent3] handle goal g5: coffee(cooked) UNTIL_END 08.15 to 08.25
44 18:50:18.969 [GoalManagerTopicAgent] scenario goal solved received! goalId: g4
45 18:50:18.998 [GoalManagerTopicAgent] sending GOAL_PLANNED : g8: documents(served) FROM_START
46 18:50:19.006 [GoalAgent1c] consume GOAL_SOLVED and end handling for goal g4: human(is_dressed) UNTIL_END
47 18:50:19.122 [GoalAgent4] consume GOAL_PLANNED : g8: documents(served) FROM_START
48 18:50:19.130 [GoalManagerTopicAgent] sending GOAL_PLANNED : g6: human(has_had_breakfast) UNTIL_END
49 18:50:19.146 [GoalAgent4] added dependencies Map(g4 -> None: --)
50 18:50:19.146 [GoalAgent4] consume GOAL_TIMEBOUNDARIES from g4: human(is_dressed) UNTIL_END 08.15 to 08.25
51 18:50:19.147 [GoalAgent4] dependencies updated Map(g4 -> g4: human(is_dressed) UNTIL_END 08.15 to 08.25)
52 18:50:19.221 [GoalAgent1d] consume GOAL_PLANNED : g6: human(has_had_breakfast) UNTIL_END
53 18:50:19.330 [GoalAgent1d] added dependencies Map(g5 -> None: --)
54 18:50:19.331 [GoalAgent1d] consume GOAL_TIMEBOUNDARIES from g5: coffee(cooked) UNTIL_END 08.15 to 08.25
55 18:50:19.331 [GoalAgent1d] dependencies updated Map(g5 -> g5: coffee(cooked) UNTIL_END 08.15 to 08.25)
56 18:50:20.954 [GoalAgent4] timeboundaries calculated 08.25 to 09.30
57 18:50:20.985 [GoalAgent1d] timeboundaries calculated 08.30 to 08.50
58 18:50:22.425 [GoalAgent1d] handle goal g6: human(has_had_breakfast) UNTIL_END 08.30 to 08.50
59 18:50:22.546 [GoalAgent4] handle goal g8: documents(served) FROM_START 08.25 to 09.30
60 18:50:48.681 [GoalManagerTopicAgent] scenario goal solved received! goalId: g5
61 18:50:48.736 [GoalAgent3] consume GOAL_SOLVED and end handling for goal g5: coffee(cooked) UNTIL_END
62 18:51:53.234 [GoalManagerTopicAgent] scenario goal solved received! goalId: g8
63 18:51:53.291 [GoalAgent4] consume GOAL_SOLVED and end handling for goal g8: documents(served) FROM_START
64 18:52:04.579 [GoalManagerTopicAgent] scenario goal solved received! goalId: g6
65 18:52:04.627 [GoalAgent1d] consume GOAL_SOLVED and end handling for goal g6: human(has_had_breakfast) UNTIL_END
66 18:52:04.639 [GoalManagerTopicAgent] sending GOAL_PLANNED : g7: human(knows_day_s_schedule) UNTIL_END
67 18:52:04.691 [GoalAgent1e] consume GOAL_PLANNED : g7: human(knows_day_s_schedule) UNTIL_END
68 18:52:04.695 [GoalAgent1e] added dependencies Map(g6 -> None: --)
69 18:52:04.737 [GoalAgent1e] consume GOAL_TIMEBOUNDARIES from g6: human(has_had_breakfast) UNTIL_END 08.30 to 08.50
70 18:52:04.737 [GoalAgent1e] dependencies updated Map(g6 -> g6: human(has_had_breakfast) UNTIL_END 08.30 to 08.50)
71 18:52:06.281 [GoalAgent1e] timeboundaries calculated 08.30 to 09.30
72 18:52:08.050 [GoalAgent1e] handle goal g7: human(knows_day_s_schedule) UNTIL_END 08.30 to 09.30
73 18:52:16.929 [GoalManagerTopicAgent] scenario goal solved received! goalId: g7
74 18:52:16.993 [GoalAgent1e] consume GOAL_SOLVED and end handling for goal
                               g7: human(knows_day_s_schedule) UNTIL_END
75
```

### D.3.2. : Versuch 2

In diesem Versuch wurde die Reihenfolge der erfüllten Ziele verändert. Vor *g4* wurde *g5* als erfüllt gemeldet. Der Ablauf verhält sich dennoch stabil.

Konsolidiertes Console-Log:

```
1 11:26:15.621 [GoalManagerTopicAgent] scenario selected received!
2 11:26:15.676 [GoalManagerTopicAgent] sending GOAL_PLANNED: g2a: homemaker(list_of_meals_composed) FROM_START
3 11:26:15.905 [GoalManagerTopicAgent] sending GOAL_PLANNED: g3b: human(shopping_is_done) UNTIL_END
4 11:26:15.956 [GoalManagerTopicAgent] sending GOAL_PLANNED: g1a: calendar(appointment_type_assigned) UNTIL_END
5 11:26:16.101 [GoalManagerTopicAgent] sending GOAL_PLANNED: g3a: homemaker(shopping_list) FROM_START
6 11:26:16.120 [GoalManagerTopicAgent] sending GOAL_PLANNED: g2: human(choosen_meal) UNTIL_END
```

## D. Tests und Versuchsaufbau

```

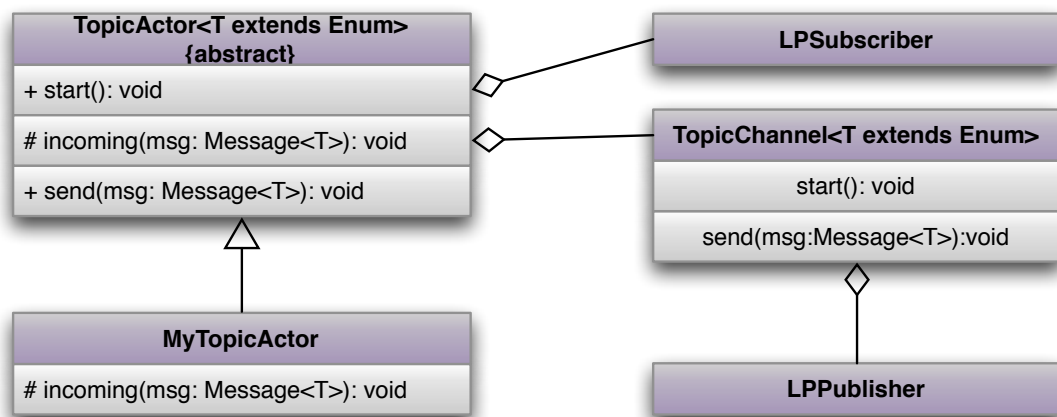
7 11:26:16.139 [GoalManagerTopicAgent] sending GOAL_PLANNED: g3: homemaker(stock_of_food_assured) EXACTLY_TO_END
8 11:26:16.160 [GoalManagerTopicAgent] sending GOAL_PLANNED:
9                                     g1: calendar(visitation_appointment_inscribed) FROM_START
10 11:26:16.956 [GoalAgent3b] consume GOAL_PLANNED: g3b: human(shopping_is_done) UNTIL_END
11 11:26:16.958 [GoalAgent2a] consume GOAL_PLANNED: g2a: homemaker(list_of_meals_composed) FROM_START
12 11:26:17.087 [GoalAgent2] consume GOAL_PLANNED: g2: human(choosen_meal) UNTIL_END
13 11:26:17.097 [GoalAgent1a] consume GOAL_PLANNED: g1a: calendar(appointment_type_assigned) UNTIL_END
14 11:26:17.112 [GoalAgent3] consume GOAL_PLANNED: g3: homemaker(stock_of_food_assured) EXACTLY_TO_END
15 11:26:17.216 [GoalAgent3] added dependencies Map(g2 → None: --)
16 11:26:17.217 [GoalAgent3a] consume GOAL_PLANNED: g3a: homemaker(shopping_list) FROM_START
17 11:26:17.236 [GoalAgent2] added dependencies Map(g1a → None: --)
18 11:26:17.242 [GoalAgent1] consume GOAL_PLANNED: g1: calendar(visitation_appointment_inscribed) FROM_START
19 11:26:17.247 [GoalAgent1] added dependencies Map()
20 11:26:18.788 [GoalAgent1] timeboundaries calculated 10.00 to 20.00
21 11:26:20.763 [GoalAgent1] handle goal g1: calendar(visitation_appointment_inscribed) FROM_START 10.00 to 20.00
22 11:26:20.814 [GoalAgent1a] consume SUBGOAL_PLANNED: g1a: calendar(appointment_type_assigned) UNTIL_END
23 11:26:20.878 [GoalAgent1a] added dependencies Map(g1 → None: --)
24 11:26:20.895 [GoalAgent1a] consume GOAL_TIMEBOUNDARIES from g1: calendar(visitation_appointment_inscribed)
25                                     FROM_START 10.00 to 20.00
26 11:26:20.895 [GoalAgent1a] dependencies updated Map(g1 → g1: calendar(visitation_appointment_inscribed)
27                                     FROM_START 10.00 to 20.00)
28 11:26:21.896 [GoalAgent1a] timeboundaries calculated 11.00 to 12.00
29 11:26:23.302 [GoalAgent1a] handle goal g1a: calendar(appointment_type_assigned) UNTIL_END 11.00 to 12.00
30 11:26:23.387 [GoalAgent2] consume GOAL_TIMEBOUNDARIES from g1a: calendar(appointment_type_assigned)
31                                     UNTIL_END 11.00 to 12.00
32 11:26:23.387 [GoalAgent2] dependencies updated Map(g1a → g1a: calendar(appointment_type_assigned)
33                                     UNTIL_END 11.00 to 12.00)
34 11:26:25.105 [GoalAgent2] timeboundaries calculated 12.00 to 13.00
35 11:26:26.580 [GoalAgent2] handle goal g2: human(choosen_meal) UNTIL_END 12.00 to 13.00
36 11:26:26.685 [GoalAgent3] consume GOAL_TIMEBOUNDARIES from g2: human(choosen_meal) UNTIL_END 12.00 to 13.00
37 11:26:26.688 [GoalAgent3] dependencies updated Map(g2 → g2: human(choosen_meal) UNTIL_END 12.00 to 13.00)
38 11:26:26.691 [GoalAgent2a] consume SUBGOAL_PLANNED: g2a: homemaker(list_of_meals_composed) FROM_START
39 11:26:26.713 [GoalAgent2a] added dependencies Map(g2 → None: --, g1a → None: --)
40 11:26:26.749 [GoalAgent2a] consume GOAL_TIMEBOUNDARIES from g2: human(choosen_meal) UNTIL_END 12.00 to 13.00
41 11:26:26.749 [GoalAgent2a] dependencies updated Map(g2 → g2: human(choosen_meal) UNTIL_END 12.00 to 13.00,
42                                     g1a → None: --)
43 11:26:26.773 [GoalAgent2a] consume GOAL_TIMEBOUNDARIES from g1a: calendar(appointment_type_assigned)
44                                     UNTIL_END 11.00 to 12.00
45 11:26:26.773 [GoalAgent2a] dependencies updated Map(g2 → g2: human(choosen_meal) UNTIL_END 12.00 to 13.00,
46                                     g1a → g1a: calendar(appointment_type_assigned) UNTIL_END 11.00 to 12.00)
47 11:26:28.270 [GoalAgent3] timeboundaries calculated 14.00 to 18.00
48 11:26:28.463 [GoalAgent2a] timeboundaries calculated 12.00 to 12.30
49 11:26:30.222 [GoalAgent3] handle goal g3: homemaker(stock_of_food_assured) EXACTLY_TO_END 14.00 to 18.00
50 11:26:30.312 [GoalAgent3a] consume SUBGOAL_PLANNED: g3a: homemaker(shopping_list) FROM_START
51 11:26:30.322 [GoalAgent3a] added dependencies Map(g3 → None: --)
52 11:26:30.324 [GoalAgent3b] consume SUBGOAL_PLANNED: g3b: human(shopping_is_done) UNTIL_END
53 11:26:30.415 [GoalAgent2a] handle goal g2a: homemaker(list_of_meals_composed) FROM_START 12.00 to 12.30
54 11:26:30.491 [GoalAgent3b] added dependencies Map(g3 → None: --, g3a → None: --)
55 11:26:30.492 [GoalAgent3b] consume GOAL_TIMEBOUNDARIES from g3: homemaker(stock_of_food_assured)
56                                     EXACTLY_TO_END 14.00 to 18.00
57 11:26:30.492 [GoalAgent3b] dependencies updated Map(g3 → g3: homemaker(stock_of_food_assured)
58                                     EXACTLY_TO_END 14.00 to 18.00,
59                                     g3a → None: --)
60 11:26:30.495 [GoalAgent3a] consume GOAL_TIMEBOUNDARIES from g3: homemaker(stock_of_food_assured)
61                                     EXACTLY_TO_END 14.00 to 18.00
62 11:26:30.495 [GoalAgent3a] dependencies updated Map(g3 → g3: homemaker(stock_of_food_assured)
63                                     EXACTLY_TO_END 14.00 to 18.00)
64 11:26:31.913 [GoalAgent3a] timeboundaries calculated 14.00 to 14.30
65 11:26:33.121 [GoalAgent3a] handle goal g3a: homemaker(shopping_list) FROM_START 14.00 to 14.30
66 11:26:33.146 [GoalAgent3b] consume GOAL_TIMEBOUNDARIES from g3a: homemaker(shopping_list)
67                                     FROM_START 14.00 to 14.30
68 11:26:33.146 [GoalAgent3b] dependencies updated Map(g3 → g3: homemaker(stock_of_food_assured)
69                                     EXACTLY_TO_END 14.00 to 18.00,
70                                     g3a → g3a: homemaker(shopping_list)
71                                     FROM_START 14.00 to 14.30)
72 11:26:34.576 [GoalAgent3b] timeboundaries calculated 15.00 to 17.00
73 11:26:36.391 [GoalAgent3b] handle goal g3b: human(shopping_is_done) UNTIL_END 15.00 to 17.00

```

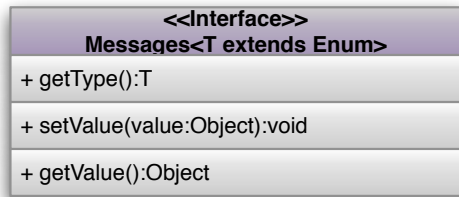
## E. ActiveMQ Actors

### E.1. ActiveMQ Actors UML

Für den Versuchsaufbau wurde ein rudimentäres Actor-Framework für die Anbindung an die ActiveMQ entwickelt. Dieses hat folgendes UML-Design:



Das Message-Interface und die Factory für Messages:



## E.2. ActiveMQ Actors Quellcode

Der Scala-Quellcode für TopicActor:

```

1 package org.livingplace.workflow.messaging
2
3 import scala.actors.Actor
4
5 import scala.actors.Actor._
6 import scala.util.parsing.json._
7 import de.hawhamburg.livingplace.messaging.activemq.wrapper._
8 import org.livingplace.workflow.messaging.MessageTypes._
9
10 /**
11  * @author Sven Tennstedt
12  *
13  */
14 class TopicActor[MType <: AbstractType](topic: String, types: MessageTypes) extends Actor {
15   val agentName = this.getClass().getSimpleName()
16   /**
17    * For receiving messages from the message bus
18    */
19   private[this] var messageReceiver: TopicActor.MessageBusReceiver[MType] = null
20
21   private[this] var topicChannel: TopicChannel[MType] = null
22
23   def act() {
24     messageReceiver = new TopicActor.MessageBusReceiver[MType](topic, self, types)
25     messageReceiver.start
26     topicChannel = new TopicChannel(topic, types)
27     topicChannel.start
28   }
29
30   def send(msg: Message[MType]) = {
31     topicChannel.send(msg)
32   }
33
34   def printMessage(msg: String): Unit = {
35     println("[ " + agentName + " ]_" + msg)
36   }

```



```
37 }
38 }
39
40 object TopicActor {
41   class MessageBusReceiver[MType <: AbstractType]
42     (topic: String, mainActor: Actor, types: MessageTypes)
43     extends Actor {
44
45     def act() {
46       loop {
47         val lpSubscriber: LPSubscriber = new LPSubscriber(topic)
48         loop {
49           val msg: String = lpSubscriber.subscribeBlocking()
50           val message: Message[MType] = parseMsg(msg)
51           if (message != null) {
52             mainActor ! message
53           }
54         }
55       }
56     }
57
58     def parseMsg(msg: String): Message[MType] = {
59       val parsed: Option[Any] = JSON.parseFull(msg)
60       if (parsed.isEmpty) {
61         return null
62       }
63       val map_message: Map[String, Any] =
64         parsed.first match {
65           case m: Map[String, Any] => m.asInstanceOf[Map[String, Any]]
66           case _                    => null
67         }
68
69       var message: Message[MType] = null
70       if (map_message != null) {
71         val str_message_type: String = map_message("message_type").asInstanceOf[String]
72
73         val message_type: MType = types.valueOf(str_message_type)
74         message = new Message[MType](message_type)
75         // message.data = map_message("value").asInstanceOf[Map[String, Any]]
76         message.data = map_message
77       }
78       return message
79     }
80 }
81 }
```

### Der Scala-Quellcode für TopicChannel:

```
1 package org.livingplace.workflow.messaging
2
3 import scala.actors.Actor
4
5 import scala.actors.Actor._
6 import scala.util.parsing.json._
7 import de.hawhamburg.livingplace.messaging.activemq.wrapper._
8 import org.livingplace.workflow.messaging.MessageTypes._
9
10 /**
11  * @author Sven Tennstedt
12  *
13  */
14 class TopicChannel[MType <: AbstractType](topic: String, types: MessageTypes) {
15   var publisher: LPPublisher = null
16   var started: Boolean = false;
17
18   def start = {
19     publisher = new LPPublisher(topic)
20     started = true

```

## E. ActiveMQ Actors

---

```
21 }
22
23 def send(msg: Message[MType]) = {
24   if (started) {
25     var json: String = Message.convertToString(msg)
26     publisher.publish(json)
27   }
28 }
29 }
```

# F. Korrespondenz

## F.1. Email Mathieu Vallée

Subject: Re: Your PhD Thesis  
From: Mathieu Vallée  
To: Sven Tennstedt

Dear Sven,

Thanks for your interest in our work. [...]

I can still answer on some points:

- first of all, our approach is mostly based on the dynamic reconfiguration of a component-based system. Although we tried a workflow based approach at first, we quickly abandoned this direction: it would not easily support adaptation to continuous context changes (as we face in our applications). Instead of planning a workflow and executing it, our agents define a meaningful configuration of available components, connect them together, then revise this configuration upon context changes (or even changes in the users needs).
- for the multi-agent system, we chose Jade, with the Jade Semantic Add-on (JSA). Although this solution does not have advanced logic-based reasoning, it was enough for us. Also Jade and the JSA are supported and partially developed by the company I was working for [...].
- in the final architecture, we have a multi-agent system with 3 main types of agents. Some of them are always running, some are activated when an application starts (there can be several applications running simultaneously). For each applications, a small team of agents is organized for managing the configuration of the application. Some agents actually belong to multiple teams, therefore taking into account the interactions and conflicts between applications. Regarding the internal agent architecture, they mostly have a knowledge base implemented with the Jena RDF library (+ OWL reasoning) and a set of procedures implemented in Java. We use simple mechanisms from Jade and the JSA to choose which procedure to execute when receiving a message or when sensing changes in the environment.

## *F. Korrespondenz*

---

- The final system worked to our satisfaction, because we designed it so :-)  
I am not sure about the exact meaning of your question. We did not face too many difficulties or unexpected behaviors, but this is mostly because we tried to keep the system simple enough. Especially, we could easily have had performance problems, but we took care of limiting the amount of messages and of reasoning to what was necessary.

[...]

Best regards,  
Mathieu Vallée

*Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.*

Hamburg, 27. Oktober 2011 Sven Tennstedt