Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Systematischer Entwurf von szenariobasierten Spezifikationen

Bachelorarbeit

im Studiengang Informatik

von

Luca Fülbier

Prüfer: Prof. Dr. Greenyer
Zweitprüfer: Prof. Dr. Schneider
Betreuer: Dipl.-Inf. Daniel Gritzner

Hannover, 23.08.2017

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 23.08.2017

Luca Fülbier

Zusammenfassung

Das Erstellen einer Anforderungsspezifikation für ein neues System ist schwer. Nicht selten erheben informelle Methoden im Requirements Engineering unvollständige oder fehlerhafte Anforderungen. Diese Fehler finden sich dann auch in den daraus abgeleiteten Spezifikationen wieder.

Formale Methoden, wie die an dem Fachgebiet Software Enginnering der Leibniz Universität enwickelten szenariobasierten Spezifikationen, helfen dem Requirements Engineer dabei, diese Probleme zu überwinden. Das Fachgebiet entwickelt außerdem ScenarioTools, eine Sammlung von Eclipse Plugins, mit denen es möglich ist, eine szenariobasierte Spezifikation direkt in Eclipse zu entwerfen und zu testen. Jedoch ist das Entwerfen einer szenariobasierten Spezifikation, gerade für Anfänger, schwer. Es fehlt eine systematische Herangehensweise, um eine szenariobasierte Spezifikation für ein neues System zu erstellen.

Zu diesem Zweck wird in dieser Arbeit ein Ansatz für den systematischen Entwurf einer szenariobasierten Spezifikation vorgestellt. Für die Aufbereitung der Informationen aus der Elicitation Phase wird KAOS, ein modellgetriebenes, Goal-orientiertes Framework genutzt. Die szenariobasierte Spezifikation wird im Anschluss an den Modellierprozess mit KAOS systematisch aus den erarbeiteten Modellen abgeleitet. Um die Praxistauglichkeit des präsentierten Ansatzes zu demonstrieren, wird der Modellierungsprozess mit KAOS sowie das Ableiten einer szenariobasierten Spezifikation für ein Fallbeispiel aus dem Bereich Home Automation durchgeführt.

Abstract

Constructing a requirements specification for a new system is hard. Informal RE methods frequently result in incomplete or incorrect requirements. These defects can then also be found in the derived specification.

Formal methods, like the scenario-based specifications, developed by the Institute for Software Enginnering at the Leibniz University Hanover, helps the requirements engineer to overcome these problems. The institute also develops ScenarioTools, a collection of Eclipse plugins, that allow creating and testing scenario-based specifications directly in Eclipse. But designing a scenario-based specification is difficult, especially for beginners. We are missing a systematic approach for designing a scenario-based specification.

For that purpose, this thesis presents an approach to systematically design a scenario-based specification. The information gathered in the elicitation phase is elaborated and structured using KAOS, a model-driven, goal-oriented RE framework. The scenario-based specification is then derived from the models that were created in the KAOS modeling process. To demonstrate the real-world use of the presented approach, the modeling process with KAOS, as well as the deduction of the scenario-based specification is done for a case study in the field of home automation.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1
1.2 Struktur der Arbeit 2

2 Grundlagen **3**
2.1 Requirements Engineering 3
2.1.1 Anforderungen 3
2.1.2 Requirements Lifecycle 4
2.2 Goal-oriented Requirements Engineering 5
2.2.1 Goals und Agenten 6
2.2.2 Goal Types 6
2.2.3 KAOS 7
2.3 Szenariobasierte Spezifikationen 7
2.3.1 ScenarioTools und SML 8
2.3.2 Simulation und Synthese 8

3 KAOS **11**
3.1 Überblick 11
3.2 Fallbeispiel: Home Automation 12
3.3 Goal Model 14
3.3.1 Goal Diagram 14
3.3.2 Erstellen des Goal Diagrams 18
3.4 Object Model 22
3.4.1 Konzeptuelle Objekte 23
3.4.2 Typen von Konzeptuellen Objekten 23
3.4.3 Grafische Darstellung 23
3.4.4 Erstellen des Object Models 25
3.5 Responsibility Model 26
3.5.1 Agenten und Agententypen 27
3.5.2 Agent Diagram 27
3.5.3 Erstellen des Agent Diagrams 28
3.6 Operation Model 30
3.6.1 Operationen und Operationstypen 30

3.6.2 Operationalization Diagram 31
3.6.3 Erstellen des Operationalization Diagrams 32
3.7 Zusammenfassung 34

4 Szenariobasierte Spezifikation 35
4.1 ScenarioTools – Überblick 35
4.2 Erstellen des Klassenmodells 36
4.3 Erstellen der Dynamischen Instanz des System Containers . . 39
4.4 SML Grundlagen 39
4.5 Erstellen der SML Spezifikation 42
4.6 Simulation und Synthese der Spezifikation 45
4.7 Zusammenfassung 46

5 Weiterführende Techniken 47
5.1 Ubiquitous Language 47
5.2 Actor Model 48

6 Diskussion 51
6.1 Bewertung des Ansatzes 51
6.2 Verbesserungvorschläge 52
6.3 Verwandte Arbeiten 53
6.3.1 Alternativen zu KAOS 53
6.3.2 Alternativen zu ScenarioTools 54
6.4 Fazit 54

7 Zusammenfassung und Ausblick 55
7.1 Zusammenfassung 55
7.2 Ausblick 56

A KAOS Modelle 57
A.1 Goal Model 57
A.2 Object Model 61
A.3 Responsibility Model 62
A.4 Operation Model 65

B Szenariobasierte Spezifikation 71
B.1 Klassenmodell 71
B.2 Objektmodell 74
B.3 SML Spezifikation 75
B.4 Runconfiguration 78

Kapitel 1

Einleitung

1.1 Motivation

Das Erstellen von Anforderungsspezifikationen für moderne Software Systeme ist ein komplizierter Prozess. Unklar formulierte Kundenwünsche und fehlendes Domänenwissen der Requirements Engineers sind potentielle Quellen für fehlerhafte Anforderungen. Nicht selten enthält die resultierende Spezifikation lückenhafte oder gar widersprüchige Anforderungen.

Das Fachgebiet Software Engineering der Leibniz Universität Hannover hat im Rahmen ihrer Forschung einen modellbasierten Ansatz, die *szenariobasierten Softwarespezifikationen*, entwickelt, welche dem Benutzer ein einfaches, formales System bieten, Objekte und Verhaltensabläufe für reaktive Systeme zu spezifizieren [14]. Ebenfalls stellt das Fachgebiet eine Sammlung von Eclipse-Plugins, namens *ScenarioTools*, zur Verfügung, mit denen szenariobasierte Softwarespezifikationen verfasst, simuliert und auf ihre Korrektheit überprüft werden können.

Diese Werkzeuge ermöglichen es zwar, vollständige und fehlerfreie Spezifikationen zu entwerfen, jedoch ist der Einstieg in die Arbeit mit dem Tool, gerade für Anfänger, schwierig. Es fehlt eine systematische Herangehensweise, um eine szenariobasierte Spezifikation für ein neues System zu erstellen.

Im Rahmen dieser Arbeit werden wir uns deshalb damit beschäftigen, wie man eine szenariobasierte Spezifikation systematisch erstellen kann.

Um das Wissen über die Domäne sinnvoll zu strukturieren, benutzen wir *KAOS*, ein Framework aus dem *Goal-oriented Requirements Engineering*, welches die Ziele, die das zukünftige System erfüllen soll, als Modellierwerkzeug nutzt, um die Anforderungen an die Software des Systems, sowie die Annahmen and die Umwelt zu identifizieren.

Aus den Modellen, welche während der Modellierung mit dem KAOS Framework entstehen, soll daraufhin eine szenariobasierte Spezifikation abgeleitet werden. Wir benutzen in dieser Arbeit ScenarioTools für das

Erstellen des Datenmodells und der Spezifikation.

Um die Praxistauglichkeit des Ansatzes zu zeigen, soll im Laufe der Arbeit das Modellieren mit KAOS, sowie das Erstellen der szenariobasierten Spezifikation anhand eines Fallbeispiels aus dem Bereich Home Automation demonstriert werden.

1.2 Struktur der Arbeit

Die Arbeit ist in mehrere aufeinander aufbauende Kapitel untergliedert. Nach einem kurzen Grundlagenkapitel beginnt der Hauptteil der Arbeit, der sich mit dem Ableiten einer szenariobasierten Spezifikation aus einem KAOS System Model beschäftigt. Danach werden kurz zwei weiterführende Techniken behandelt und das im Hauptteil vorgestellte Verfahren diskutiert. Es folgt eine kurze Erklärung des Inhalts der einzelnen Kapitel.

Kapitel 2 vermittelt Grundlagenwissen über die unterschiedlichen Phasen des Requirements Engineering, das Goal-oriented Requirements Engineering, sowie szenariobasierte Spezifikationen.

Kapitel 3 beschäftigt sich mit dem KAOS Framework und erläutert die verschiedenen Teilmodelle anhand eines Fallbeispiel aus dem Bereich Home Automation.

Kapitel 4 erstellt aus den erarbeiteten KAOS Modellen systematisch eine szenariobasierte Spezifikation in ScenarioTools.

Kapitel 5 präsentiert weiterführende Techniken, die dazu genutzt werden können, um den Schritt von einer szenariobasierten Spezifikation zu der Implementation des Systems zu erleichtern.

Kapitel 6 diskutiert den in der Arbeit präsentierten Ansatz zum systematischen Erstellen einer szenariobasierten Spezifikation und zeigt Vor- und Nachteile der benutzten Werkzeuge. Abschließend werden einige verwandte Arbeiten präsentiert und ein Fazit gezogen.

Kapitel 2

Grundlagen

In diesem Kapitel wird das Grundlagenwissen vermittelt, welches für den Hauptteil der Arbeit vorausgesetzt wird. Im Detail werden die unterschiedlichen Anforderungstypen und Phasen des *Requirements Engineering* betrachtet, das *Goal-oriented Requirements Engineering* und seine grundlegenden Bausteine, sowie die *Szenariobasierten Spezifikationen*, welche unser Spezifikationswerkzeug für Kapitel 4 sein werden.

2.1 Requirements Engineering

Requirements Engineering ist der Bereich des Software Engineering, der sich damit befasst, die Anforderungen und Einschränkungen an ein Software System zu erheben, zu präzisieren und schriftlich festzuhalten [25].

2.1.1 Anforderungen

Das Requirements Engineering ist ein interdisziplinärer Prozess [25], der die anderen Bereiche des Software Engineerings maßgeblich beeinflusst, indem funktionale und nicht-funktionale Anforderungen zusammen mit den Stakeholdern erhoben werden, welche die Informationsbasis für nachfolgende SE-Prozesse bilden.

Funktionale Anforderungen machen eine Aussage darüber, welche Funktionen das System erfüllen muss [6]. Diese Anforderungen beschreiben dabei im Detail, welche Operationen von dem System automatisiert werden sollen und unter welchen Umständen [21].

Nicht-funktionale Anforderungen bilden eine größere Gruppe von Anforderungen, welche Aussagen über Performance-Ziele und andere Qualitätsmerkmale des Software Systems machen [6]. Neben den Performance-Zielen (Reaktionszeit, Durchsatz, Speicherbedarf) sind besonders die Sicherheits-Ziele (engl. *Security and Safety Requirements*) zu erwähnen, die für sicherheitskritische Systeme, wie Software für medizinische

Geräte, eine große Rolle spielen.

Neben den oben aufgeführten Anforderungen gibt es außerdem noch eine Menge weiterer Anforderungen, die je nach Definition eine eigene Gruppe bilden oder den nicht-funktionalen Anforderungen zugeordnet werden. Dazu gehören Regelkonformitäten (Gesetze, kulturelle Einflüsse, politische Einflüsse), Architekturelle Anforderungen (Zielsystemeinschränkungen, Geographische Verteilung) und Anforderungen an die Entwicklung (Kostengrenzen, Deadlines, Wartbarkeit, Dokumentation, usw.) [21].

Ein **Stakeholder** ist eine Person oder eine Gruppe, die durch die zu bauende Software beeinflusst wird oder Einfluss auf den Prozess haben kann [18]. Zu diesen Personen(gruppen) gehören zum Beispiel der Kunde, die Benutzer der Software sowie die Entwickler der Software.

Eine vernünftige Analyse der funktionalen und nicht-funktionalen Anforderungen ist für den Erfolg eines Projektes äußerst wichtig. Eine Studie der Standish Group im Jahr 1994 zeigt, dass der am häufigsten aufgeführte Grund für den Misserfolg von Software Projekten mit 13.1% aller abgebrochenen Projekte unvollständige Anforderungen sind. An zweiter Stelle folgt die mangelnde Einbeziehung der Benutzer mit 12.4% (ebenfalls ein Bereich, mit dem sich das Requirements Engineering befasst) und erst an dritter Stelle fehlende Ressourcen mit 10.6% [19].

2.1.2 Requirements Lifecycle

Der Requirements Lifecycle beschreibt die unterschiedlichen Aktivitäten, die während des RE-Prozesses durchgeführt werden, um Anforderungen zu erheben, auszuwerten und in einer Spezifikation festzuhalten. Die folgende Einteilung der Aktivitäten basiert auf der Forschung von Axel van Lamsweerde [21], einem belgischen Professor, der ebenfalls die KAOS-Methode entwickelt hat, welche in Kapitel 3 vorgestellt wird.

Das Verstehen der Domäne (engl. ***Domain Understanding***) geschieht vor allen anderen Aktivitäten des RE. Das Ziel ist es, mit der Domäne vertraut zu werden und grundlegende Probleme aufzudecken. Domänenwissen kann aus einer Vielzahl von Quellen bezogen werden, darunter Dokumenten- und Buchrecherche, Untersuchung von ähnlichen Systemen sowie Gespräche mit Domänenexperten und Beobachtung von Stakeholdern.

Das Herauslocken von Anforderungen (engl. ***Requirements Elicitation***), beschäftigt sich mit dem Erheben von Anforderungen an das System, welches gebaut werden soll (*System-to-be*). Dies geschieht, indem Stärken und Schwächen des existierenden Systems (*System-as-is*) identifiziert werden und, mit diesem Hintergrundwissen Fragen über das System-to-be gestellt werden. Der Begriff System-as-is ist dabei nicht auf ein bestehendes Software Produkt beschränkt, sondern kann auch einen manuellen Prozess beschreiben, der durch das System-to-be automatisiert werden soll. Das Erheben von Anforderungen ist ein explorativer Prozess und sollte in enger

Kooperation mit den Stakeholdern durchgeführt werden, um einen möglichst vollständigen Anforderungskatalog zu erhalten.

Die Ergebnisse des Domain Understandings und der Requirements Elicitation Phase werden typischerweise in einem vorläufigen unstrukturierten Spezifikationsdokument festgehalten und im nächsten Schritt ausgewertet und verhandelt (engl. ***Evalutation and Agreement***). Ziel ist es, Konflikte in den Anforderungen aufzulösen, Risiken zu identifizieren und zu behandeln, die beste Option aus Alternativen zu wählen und Anforderungen zu priorisieren. Auch diese Erkenntnisgewinne sollten in der vorläufigen Spezifikation niedergeschrieben werden.

Das vorläufige Spezifikationsdokument wird im Spezifikations- und Dokumentations-Schritt (engl. ***Specification and Documentation***) strukturiert und auf den Punkt gebracht. Das Produkt dieser Aktivität wird *Requirements Document* genannt und unterliegt keiner bestimmten Form. Es gibt eine Vielzahl von Spezifikations-Templates und Methoden, welche genutzt werden können, um diesen Arbeitsschritt zu erleichtern. Die Szenariobasierten Spezifikationen bieten eben so eine Methode. Der Hauptteil dieser Arbeit geht näher darauf ein, wie eine solche Szenariobasierte Spezifikation systematisch erstellt werden kann.

Die letzte Aktivität im Requirements Lifecycle ist das Verfestigen der Anforderungen (engl. ***Requirements Consolidation***), wobei die Qualität des Anforderungsdokuments aus dem vorhergegangengen Schritt überprüft wird, die Anforderungen zusammen mit den Stakeholdern validiert (Erfüllt das spezifizierte System alle Wünsche der Stakeholder?) und untereinander verifiziert werden (Ist das spezifizierte System konsistent und vollständig?).

Wie der Name Requirements Lifecycle schon andeutet, bilden die oben erläuterten Aktivitäten einen iterativen Prozess. Neue Erkenntnisse, Unklarheiten und der Input von Stakeholdern sorgen dafür, dass das Anforderungsdokument Stück für Stück verbessert wird. Die unterschiedlichen Aktivitäten können sich dabei überspringen oder sich überlappen. Ziel ist, wie bei der iterativen Software Entwicklung, ein möglichst vollständiges und korrektes Endprodukt – in diesem Fall das Requirements Document, welches für die weitere Entwicklung des Software Systems eine zentrale Rolle spielt.

2.2 Goal-oriented Requirements Engineering

Goal-oriented Requirements Engineering ist eine Form des Requirements Engineering, in dem die Ziele (engl. *Goals*) des Systems genutzt werden, um Anforderungen zu erheben, zu verbessern und in einer Anforderungsspezifikation festzuhalten [21].

2.2.1 Goals und Agenten

Ein Ziel (engl. ***Goal***) ist eine Absichtserklärung, welche das System mit Hilfe seiner Agenten erfüllen soll [21]. Ziele können auf unterschiedlichen Abstraktionsniveaus definiert werden. High-Level Goals, wie *"Transportiere Fahrgast sicher zu seinem gewünschten Zielort"*, setzen sich dabei aus konkreteren Low-Level Goals zusammen, wie zum Beispiel *"Halte gesetzlich vorgeschriebenen Mindestabstand zu anderen Verkehrsteilnehmern ein"* [20].

Ein System besteht neben der zu entwerfenden Software außerdem noch aus seiner Umgebung (engl. *Environment*). Die Umgebung eines autonomen Fahrsystems wären zum Beispiel eine Vielzahl von Sensoren und Aktoren, andere in dem Auto verbaute Systeme, andere Verkehrsteilnehmer, der zu transportierende Fahrgast, usw.

Ein **Agent** ist ein Teilnehmer des Systems, welcher aktiven Einfluss auf den Systemverlauf nehmen kann. Typische Agenten in einem System sind Benutzer, Geräte, die aktiv Änderungen in dem System auslösen, sowie die Software, die das System steuert. Passive Elemente, wie zum Beispiel die Windschutzscheibe, sind keine Agenten [20]. Es ist anzumerken, dass die primäre Software-Komponente des zu spezifizierenden Systems ebenso ein Agent ist, wie andere Software, mit der das System integriert werden muss.

2.2.2 Goal Types

Goals werden in *Behavioural Goals* und *Soft Goals* unterschieden. Behavioural Gaols beschreiben deklarativ ein zu erreichendes Systemverhalten. Hier wird zusätzlich in *Achieve* und *Maintain* Goals unterschieden [21].

Achieve Goals beschreiben ein zu erreichendes Systemverhalten, wo die Zielbedingung früher oder später erreicht werden muss, falls eine entsprechende Ausgangsbedingung wahr ist [21]. Ein Achieve Goal für ein Fahrstuhlsystem könnte zum Beispiel sein, dass sich der Fahrstuhl zu einem Stockwerk bewegt, sobald der Fahrstuhlknopf in dem entsprechenden Stockwerk gedrückt wurde:

Achieve[MoveElevatorToFloorIfRequested]

Maintain Goals schreiben ein Systemverhalten vor, welches zu jedem Zeitpunkt der Systemausführung erfüllt sein muss [21]. Ein Maintain Goal für das Fahrstuhlsystem wäre zum Beispiel, dass die Türen des Fahrstuhls während der Fahrt geschlossen bleiben:

Maintain[DoorsClosedWhileMoving]

Im Gegensatz zu Behavioural Goals beschreiben **Soft Goals** Vorlieben über das Systemverhalten, welche auch teilweise erfüllt sein können, statt nur vollständig oder gar nicht. [21]. Für das Fahrstuhlsystem könnte zum

Beispiel gewünscht werden, dass die Wartezeit der Fahrgäste möglichst kurz gehalten wird:

Reduce[PassengerWaitTime]

Die oben genutzte *Verb*[Zielverhalten] Schreibweise wird in Kapitel 3 für die Benennung der Goals genutzt. Achieve Goals mit negierter Zielbedingung können ebenfalls durch das *Cease* Verb beschrieben werden. Maintain Gaols mit negierter Zielbedingung durch das *Avoid* Verb. Soft Goals können durch alle Verben beschrieben werden, die eine Vorliebe über das Zielverhalten ausdrücken (zB. *Reduce/Increase*, *Maximize/Minimize*) [21].

2.2.3 KAOS

KAOS ist ein Goal-orientiertes, modellbasiertes Framework zur Erhebung von Anforderungen. Das Akronym steht ursprünglich für *Knowledge Acquisition in autOmated Specification* [13], was in späteren Publikationen jedoch zu *Keep All Objects Satisfied* geändert wurde [22].

Axel Van Lamsweerde war in großem Maße an der Forschung und Entwicklung des KAOS Frameworks beteiligt, gibt dem Ansatz in seinem Buch *Requirements Engineering - From System Goals To UML Models To Sofware Specifications* [21] jedoch keinen Namen. Stattdessen widmet er den zweiten Teil seines Buches einem sogenannten *modellbasierten Ansatz für Requirements Engineering.* In dieser Arbeit wird der Ansatz aus Vam Lamsweerdes Buch präsentiert, jedoch weiterhin KAOS genannt. Diese Wahl wurde getroffen, da KAOS selbst in aktueller Forschungsliteratur ein wiederkehrender Begriff für den Ansatz ist [16].

Wir haben uns in dieser Arbeit für KAOS entschieden, da das Framework, genau wie die szenariobasierten Spezifikationen auch, das System in Software und Umwelt trennt und einen hohen Fokus auf den Informationsfluss und die Interaktionen der Systemteilnehmer legt. Kapitel 6.3 gibt eine kurze Übersicht über andere, ebenfalls Goal-orientierte RE Frameworks.

Kapitel 3 befasst sich im Detail mit den unterschiedlichen Modellen, die im Laufe des Requirement Engineerings mit KAOS entworfen werden sowie die Anwendung an einem Fallbeispiel.

2.3 Szenariobasierte Spezifikationen

Szenariobasierte Spezifikationen sind ein Spezifikationswerkzeug, welches am Fachgebiet für Software Engineering der Leibniz Universität Hannover erforscht und entwickelt wird. Eine szenariobasierte Spezifikation verfolgt dabei einen anderen Ansatz wie typische textuelle Spezifikationen. Anstatt die Anforderungen an ein Produkt Punkt für Punkt festzuhalten, beschreibt

man in einer szenariobasierten Spezifikation die Interaktionen der Systemteilnehmer. Szenariobasierte Spezifikationen eignen sich deshalb besonders gut zum Spezifizieren von Reaktiven Systemen.

2.3.1 ScenarioTools und SML

ScenarioTools [1] ist ein Tool zum Erstellen von szenariobasierten Spezifikationen, welches ebenfalls am Fachgebiet für Software Engineering der Leibniz Universität Hannover entwickelt wurde. ScenarioTools ist eine Sammlung von Eclipse Plugins, die es ermöglichen, eine szenariobasierte Spezifikation aus vier Modellen zu erstellen [9].

Das erste Modell ist das **Klassenmodell** der Domäne, welches mit Hilfe des Eclipse Modeling Framework [2] erstellt werden kann. Dieses Modell beschreibt die unterschiedlichen Objekte des zu spezifizierenden Systems, ihre Relationen sowie ihre Operationen [9].

Das zweite Modell ist das **Objektmodell**. Das Objektmodell ist aus objektorientierter Sicht eine konkrete Instanz des Klassenmodells, in dem die benötigten Teilnehmer der Domäne instanziert und initialisiert wurden [9].

Das dritte Modell ist die **SML Spezifikation**, welche in der *Scenario Modeling Language (SML)* verfasst wird. SML ist eine domänenspezifische Sprache, welche entwickelt wurde, um möglichst intuitiv das Verhalten zwischen den Instanzen des Objektmodells zu beschreiben. SML ist eine textbasierte Modellierungssprache, die sich stark an Life Sequence Charts orientiert, aber auch eigene Erweiterungen gegenüber LSCs besitzt [9].

Das vierte und letzte Modell ist die **Runconfiguration**, welches Objektmodell und SML Spezifikation mit Hilfe zusätzlicher Konfigurationsinformationen verknüpft.

2.3.2 Simulation und Synthese

Diese oben beschriebenen Modelle bilden den Kern einer szenariobasierten Spezifikation in ScenarioTools. Da die Modelle alle auf formalen Modellierungsansätzen basieren, ist ScenarioTools in der Lage, die komplette Spezifikation zu simulieren und mit Hilfe eines Synthese-Algorithmus ihre Erfüllbarkeit zu überprüfen.

Die **Simulation** erlaubt es dem Entwickler, die Spezifikation Schritt für Schritt per Hand durchzuspielen. Dieser Vorgang kann mit dem schrittweisen Durchgehen eines endlichen Automaten verglichen werden. Die Simulation ist dabei schlau genug, wiederkehrende Zustände zu bemerken und diese in einem Simulationsgraphen darzustellen [9].

Die **Controller Synthese** führt eine automatisierte Analyse der Spezifikation durch, indem der komplette Graph des Systems inklusive aller

[1] www.scenariotools.org
[2] www.eclipse.org/modeling/emf

Zustände und Transitionen generiert wird. In diesem Graph wird dann nach einer Strategie gesucht, welche alle Szenarien erfüllt. Die Existenz einer solchen Strategie bedeutet, dass die synthetisierte Spezifikation in sich konsistent und damit erfüllbar ist. Fehlerhafte Systemzustände werden in dem generierten Graphen markiert ermöglichen es dem Entwickler, den verantwortlichen Teil der Spezifikation zu identifizieren und zu korrigieren [9].

Die Benutzung von ScenarioTools und das Schreiben von Spezifikationen in SML werden in Kapitel 4 im Detail behandelt.

Kapitel 3

KAOS

In diesem Kapitel wird die Arbeit mit dem KAOS Framework anhand eines Fallbeispiels präsentiert. Als Fallbeispiel dient ein Home Automation System, welches den Wohnkomfort älterer Menschen steigern und die benötigten Heizkosten der Wohnung senken soll.

3.1 Überblick

Die in diesem Kapitel präsentierten Verfahren stammen aus dem zweiten Teil des Buches *Requirements Engineering - From System Goals To UML Models To Software Specifications* [21] von Axel Van Lamsweerde. Der zweite Teil des Buches befasst sich mit dem Erstellen eines *System Model* für ein System-to-be. Das System Model ist eine abstrakte Repräsentation eines Systems, welches sich aus mehreren verknüpften Teilmodellen zusammensetzt.

Den Kern des System Models, welches mit KAOS entworfen wird, bildet das **Goal Model**. Das Goal Model zeigt die funktionalen und nicht-funktionalen Ziele des Systems sowie ihre Abhängigkeiten untereinander. Dies ermöglicht es dem Betrachter des Modells, einen groben Überblick über das System (Top-Level Goals) sowie eine detaillierte Sicht auf die Teilziele (Refinement Goals) zu erlangen. Das Goal Model wird von den Anforderungsspezialisten so lange verfeinert, bis sich Teilziele ergeben, die genau einem Agenten des Systems zugewiesen werden können. Diese Ziele werden dann *Requirements* für Software Agenten und *Expectations* für Environment Agenten genannt. Zusätzlich werden in dem Goal Model alternative Verfeinerungsziele festgehalten, die während der Implementationsphase in unterschiedlichen Systementwürfen resultieren können.

Es lassen sich weitere Teilmodelle aus dem Goal Model ableiten, welche unterschiedliche Ebenen des Requirements Engineering abdecken.

Das **Obstacle Model** zeigt potentielle Fehlerquellen in dem Goal Model auf, mit dem Ziel, Gegenmaßnahmen in Form von zusätzlichen Goals zu finden, um ein robusteres Endsystem zu erhalten.

Das **Object Model** präsentiert eine strukturelle Sicht auf die konzeptuellen Objekte des Systems. Dieses Modell ist vergleichbar mit einem UML Klassendiagramm, welches sich auf Klassen, Attribute und Assoziationen zwischen den Konzepten beschränkt.

Das **Agent Model** zeigt die Verantwortlichkeiten der Agenten innerhalb des Systems. Ziel ist es, Agentenrollen zu verfeinern und die Verantwortlichkeiten entsprechend der Kapazitäten der Agenten so zu verteilen, dass alle Ziele des System-to-be erfüllt werden können.

Das **Operation Model** zeigt eine funktionale Sicht des Systems. Die benötigten Funktionen werden als Operationen der Agenten modelliert, welche unter bestimmten Bedingungen ausgeführt werden können, um die Systemziele aus dem Goal Model zu erfüllen.

Das **Behaviour Model** präsentiert die Interaktionen zwischen den Agenten sowie die Abläufe innerhalb des Systems mit Hilfe von Szenarios und State Machines.

Im weiteren Verlauf dieses Kapitels beschäftigen wir uns damit, die unterschiedlichen Modelle für ein Fallbeispiel zu erarbeiten. Dabei gehen wir sequentiell vor, indem wir zuerst das Goal Model erstellen und danach Object Model, Agent Model und Operation Model ableiten und verfeinern. Das Obstacle Model wird nicht erstellt, da dies den Rahmen der Arbeit übersteigen würde. Das Behaviour Model wird ebenfalls nicht behandelt, da sich Kapitel 4 im Detail damit befasst, die Informationen der anderen KAOS-Modelle in eine szenariobasierte Spezifikation zu überführen, welche das Behaviour Model ersetzt.

3.2 Fallbeispiel: Home Automation

Das Arbeiten mit KAOS soll anhand eines Fallbeispieles gezeigt werden. In dieser Arbeit haben wir uns für ein Beispielsystem aus dem Bereich Home Automation entschieden. Das System soll ein gesundes Wohnklima für ältere Menschen garantieren und gleichzeitig den Energiebedarf der Heizung reduzieren.

Das Ergebnis der ersten Requirements Elicitation Phase mit der fiktiven Firma SmartHome könnte zum Beispiel so aussehen:

> **System-as-is** Es ist schwierig für alte Menschen, ein angenehmes und gesundes Klima in ihrer Wohnung zu genießen und gleichzeitig energieeffizient zu wohnen. Vor-Ort Studien zeigen, dass alte Menschen dazu tendieren, ihre Wohnung zu überheizen. Ebenfalls klagten viele der Probanden darüber, dass die Wohnung am frühen Morgen zu kalt sei. Einige Probanden umgingen dieses Problem, indem sie die komplette Nacht durch heizten.

System-to-be Die Firma SmartHome möchte dieses Problem lösen, indem sie ein zentrales, intelligentes Klimatisierungssystem in den Häusern der Kunden installiert. Das System soll dafür sorgen, dass der Wohnbereich des Kunden zu Nutzzeiten stets ein gesundes Wohnklima aufweist. Sowohl innerhalb als auch ausserhalb der Nutzzeiten soll der Energiebedarf der Klimatisierung auf ein Minimum reduziert werden. Um dies zu erreichen, wird ein spezielles Weck-System in den Schlafräumen der Kunden installiert, an dem die Weckzeit eingestellt werden kann. Eine halbe Stunde, bevor der Kunde geweckt wird, soll der Wohnraum bereits angeheizt werden. Die benötigte Heizleistung wird kontinuierlich angepasst, indem die aktuelle Temperatur über einen Temperatursensor gemessen und mit der gesundheitlich optimalen Raumtemperatur von 22°C verrechnet wird. Die Formel zur Berechnung der benötigten Heizleistung wird von den SmartHome Ingenieuren bestimmt und liegt als externe Systemkomponente vor. Neben dem Wecksystem wird außerdem ein Bewegungssensor in der Tür der Schlafzimmers des Kunden angebracht. Sollte sich der Kunde zwischen 20 und 4 Uhr länger als 30 Minuten im Schlafzimmer aufhalten, sollen alle Heizungen im Wohnraum ausgeschaltet werden, um Energie während der Nachtruhe zu sparen.

Die zweite Elicitation Runde beantwortet offene Fragen und definiert einige Begriffe:

Das intelligente Klimatisierungssystem der Firma SmartHome steuert die Beheizung aller Räume, welche bei der Installation des Systems von dem Benutzer als Wohnraum angegeben werden. Davon ausgenommen sind Schlafzimmer und Badezimmer. Diese Zimmer werden durchgängig auf 22°C geheizt, da sie zu jeder Tageszeit genutzt werden.

Sollte die Raumtemperatur 22°C oder mehr betragen, soll nicht geheizt werden. Ein Kühlsystem ist nicht vorgesehen, da Klimaanlagen und Ventilatoren sich in vergangenen Studien als gesundheitlich kontraproduktiv erwiesen haben.

Kunde Ältere Menschen, die das Haus nur selten oder nie verlassen. Das Marketing für das Klimatisierungssystem wendet sich ebenfalls an diese Zielgruppe.

Klima Wärmegrad der Wohnung. 22°C gilt als das optimale Wohnklima für einen gesunden Wohnraum.

Wohnraum Räume der Wohnung, die primär tagsüber von dem Kunden genutzt werden. Spezielle Räume wie Keller

und Abstellkammer können von dem Kunden bei der Installation vom Wohnraum ausgeschlossen werden, um den benötigten Energiebedarf weiter zu reduzieren. Das Schlafzimmer sowie das Badezimmer werden innerhalb des Systems separat identifiziert, da diese auch außerhalb der Nutzzeit beheizt werden sollen.

Nutzzeit Zeitraum, in dem der Wohnraum potentiell genutzt wird. Für das Klimatisierungssystem erstreckt sich dieser Zeitraum von der Weckzeit bis hin zur Bettruhe.

Weckzeit Zeitpunkt, zu dem der Kunde vom Wecksystem geweckt wird. Die Weckzeit kann vom Kunden am Wecksystem eingestellt werden.

Bettruhe Zeitpunkt, zu dem der Kunde schlafen geht. Dieser Zeitpunkt wird dann gesetzt, wenn sich der Kunde zwischen 20 und 4 Uhr länger als 30 Minuten im Schlafzimmer aufhält, ohne dieses zu verlassen. Sollte sich der Kunde bereits vor 20 Uhr im Schlafzimmer befunden haben, wird die verstrichene Zeit ebenfalls angerechnet. Es wird davon ausgegangen, dass der Kunde den Wohnraum nachts nicht nutzt.

Auf Basis dieser Informationen sollen nun die unterschiedlichen KAOS Modelle erstellt werden.

3.3 Goal Model

Das Goal Model ist das wichtigste Teilmodell, das während des Modellieren mit KAOS entsteht. Es zeigt, wie die funktionalen und nicht-funktionalen Ziele des Systems miteinander zusammenhängen, wie Ziele in Teilziele verfeinert werden können und welche alternativen Lösungsansätze es für bestimmte Systemfunktionen gibt.

3.3.1 Goal Diagram

Das Goal Model wird grafisch als ein AND/OR Graph dargestellt. Dieser Graph wird *Goal Diagram* genannt.

Ziele und Annahmen

Die Knoten des Graphen stellen die Ziele des Systems dar und werden in der aus dem Grundlagen-Kapitel bekannten *Verb*[Zielverhalten] Schreibweise beschrieben. Das Zielverhalten wird in CamelCase notiert und sollte möglichst aussagekräftig sein, so dass das Goal Diagram leicht

verständlich ist. Zusätzliche Informationen, wie textuelle Beschreibungen, können als Annotation direkt in dem Goal Diagram oder in einem separaten Dokument hinterlegt werden. Goal Knoten werden grafisch als ein lehnendes Parallelogramm dargestellt. Abbildung 3.1 zeigt zum Beispiel das Ziel *Maintain*[CorrectHeatingPowerCalculated], welches in mehrere Teilziele verfeinert wird.

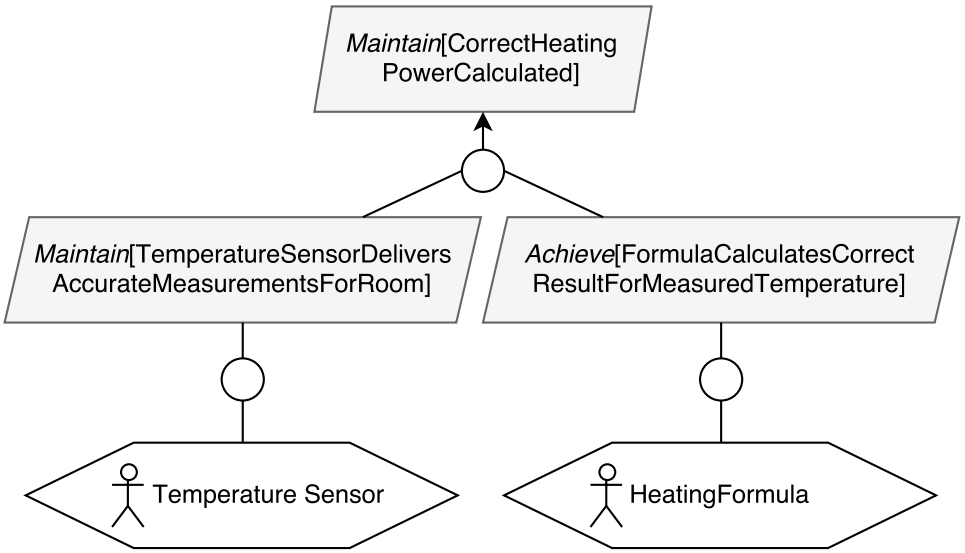

Abbildung 3.1: Goal Diagram - Ziel für das Berechnen der korrekten Heizleistung. Die Agenten *TemperatureSensor* und *HeatingFormula* sind für die Erfüllung der Teilziele verantwortlich

Eine besondere Knoten-Art bilden die Annahmen (engl. *assumptions*). Annahmen treffen eine Aussage über das System, welche wahr sein muss. Ist eine solche Annahme nicht wahr, können bestimmte Ziele des Systems nicht mehr erfüllt werden. Annahmen werden grafisch als Trapez dargestellt.

Anzumerken ist, dass die Ziele im Goal Model häufig wenige bis gar keine Vorschriften über die Implementation des Systems machen. Das *Achieve*[WakeupTimeConfiguredByUser] Ziel aus Abbildung 3.2 schreibt zum Beispiel nicht vor, wie die Benutzeroberfläche für das Einstellen der Weckzeit auszusehen hat. Dies macht das Modell robuster gegenüber Änderungen. Ein Wechsel von analogem zu digitalem Einstellen würde zum Beispiel keine Änderung an dem Goal Diagram bewirken. Sollte der Kunde allerdings implementationsnahe Anforderungen an das System haben, können diese ebenfalls modelliert werden. Ein Beispiel dafür ist das Ziel *Achieve*[FormulaCalculatesNoHeatingEnergyNeededIfAbove22Degrees] aus Abbildung 3.2, welches klar vorgibt, wie die Berechnung der Heizleistung sich verhalten soll, sobald die gemessene Temperatur über 22°C liegt.

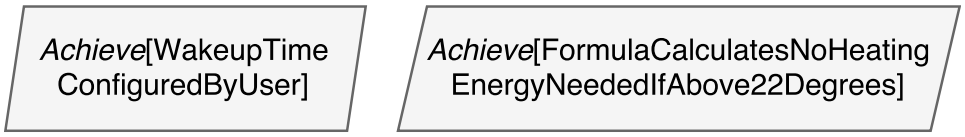

Abbildung 3.2: Goal Diagram - Ziel ohne Implementationsannahme links, Ziel mit Implementationsvorschrift rechts

AND Verknüpfungen

Die Kanten zwischen den Knoten des Graphen stellen die AND und OR Verknüpfungen dar. Laufen mehrere Kanten in einem Kreis zusammen, sind diese AND verknüpft. Das bedeutet, dass alle AND verknüpften Ziele erfüllt sein müssen, damit das verfeinerte Ziel ebenfalls erfüllt ist.

Abbildung 3.1 zeigt das Ziel des Home Automation Systems, dass die benötigte Heizleistung stets richtig berechnet wird. Dieses Ziel wurde in zwei Teilziele verfeinert. Zum einen muss der Temperatursensor eine akkurate Messung der Zimmertemperatur abliefern. Zum anderen muss die Heizformel-Systemkomponente die richtige Heizleistung für die gemessene Temperatur errechnen. Sind die beiden Ziele erreicht, ist auch das überliegende Ziel der korrekten Heizleistungsberechnung erfüllt.

Das Verfeinern von Zielen mit Hilfe von AND Verknüpfungen ist das Hauptwerkzeug für die Modellierung des Systems im Goal Diagram. Im späteren Verlauf dieses Kapitels werden Methoden vorgestellt, die es erleichtern, bestehende Ziele zu verfeinern und überliegende Ziele zu finden.

OR Verknüpfungen

Eine OR Verknüpfung wird dazu benutzt, Verfeinerungsalternativen im Goal Diagram darzustellen. Das bedeutet, dass eine einzelne Alternative (meist mehrere AND verknüpfte Ziele) ausreicht, um das verfeinerte Ziel komplett zu erfüllen. Welche der Alternativen schlussendlich im System-to-be implementiert wird, entscheidet sich im weiteren Verlauf des Requirements Engineering, wenn Vor- und Nachteile der Alternativen herausgearbeitet werden oder Alternativen aufgrund von Konflikten mit anderen Zielen nicht realisiert werden können.

Abbildung 3.3 zeigt zwei OR verknüpfte Lösungsalternativen für das Ziel, dass keine Heizleistung aufgewendet werden soll, sobald die Temperatur über 22°C beträgt. Die erste Alternative schaltet die Heizung bei einer Temperatur von 22°C oder höher ab und bei einer Temperatur von unter 22°C an. Die zweite Alternative setzt voraus, dass die aufgewendete Heizleistung bei 22°C oder mehr durchgehend Null beträgt. Dies könnte garantiert werden, indem die von den SmartHome Ingenieuren erstellte Heizleistungsformel diesen Sonderfall mit einbezieht. Beide Alternativen

benötigen ein genaues Messergebnis des entsprechenden Temperatursensors, um das verfeinerte Ziel zuverlässig zu erfüllen.

Es ist wichtig zu erwähnen, dass Verfeinerungsalternativen während des Modellierens aufgelöst werden müssen, indem Entscheidungen darüber getroffen werden, welche der Alternativen den größten Nutzen für das System mit sich bringt. Spätestens vor dem Ableiten einer Spezifikation aus den KAOS Modellen dürfen keine Alternativen mehr existieren, da dies nicht mehr zu einer eindeutigen Spezifikation führen würde.

Um die Modelle in dieser Arbeit möglichst kurz und präzise zu halten, werden wir auf Verfeinerungsalternativen in den Modellen verzichten. Es sei jedoch gesagt, dass Verfeinerungsalternativen in realen Projekten ein wichtiges Werkzeug sind, da sie dem Kunden die Möglichkeit bieten, durch die Wahl einer Alternative bereits im RE-Prozess Einfluss auf das Endsystem zu haben.

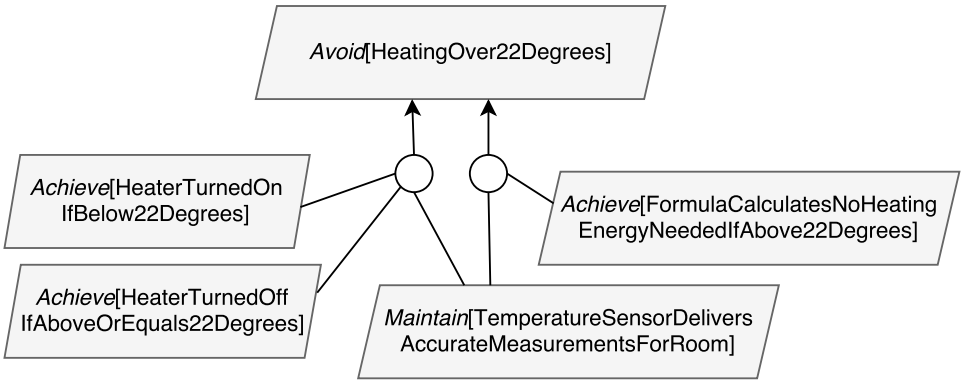

Abbildung 3.3: Goal Diagram - Verfeinergungsalternativen für die Überheizungsvermeidung

Zuweisung von Agenten

Ziel ist es, die einzelnen Goals so weit zu verfeinern, dass es möglich ist, die Verantwortlichkeit für die Erfüllung eines Leaf-Goals einem einzelnen Agenten im System zuzuweisen. Agenten werden grafisch als ein Hexagon dargestellt und durch eine einzelne AND-Verknüpfung einem Leaf-Goal zugewiesen.

Abbildung 3.1 weist den verfeinerten Zielen jeweils einen Agenten zu, der für das Erfüllen dieses Ziels im System verantwortlich ist. Der Temperatursensor ist dafür verantwortlich, ein korrektes Messergebnis abzuliefern und die Heizformel-Systemkomponente ist dafür verantwortlich, die korrekte Heizleistung für die gemessene Temperatur zu errechnen. Wir sprechen bei diesen Zielen von Expectations, da die zugewiesenen Agenen Environment Agenten sind. Die Software hat keinen Einfluss auf Expectations, kann

jedoch nur bei ihrer Erfüllung korrekt funktionieren.

Für das Abschalten aller Wohnraumheizungen ist hingegen der System Controller verantwortlich (siehe Abbildung 3.4). Solche Ziele nennen wir Requirements, da der System Controller ein Software Agent ist. Requirements werden von der Software selbst erfüllt.

Um Software Agenten von Environment Agenten zu unterscheiden, werden Environment Agenten im Goal Diagram durch ein kleines Strichmännchen im Hexagon gekennzeichnet.

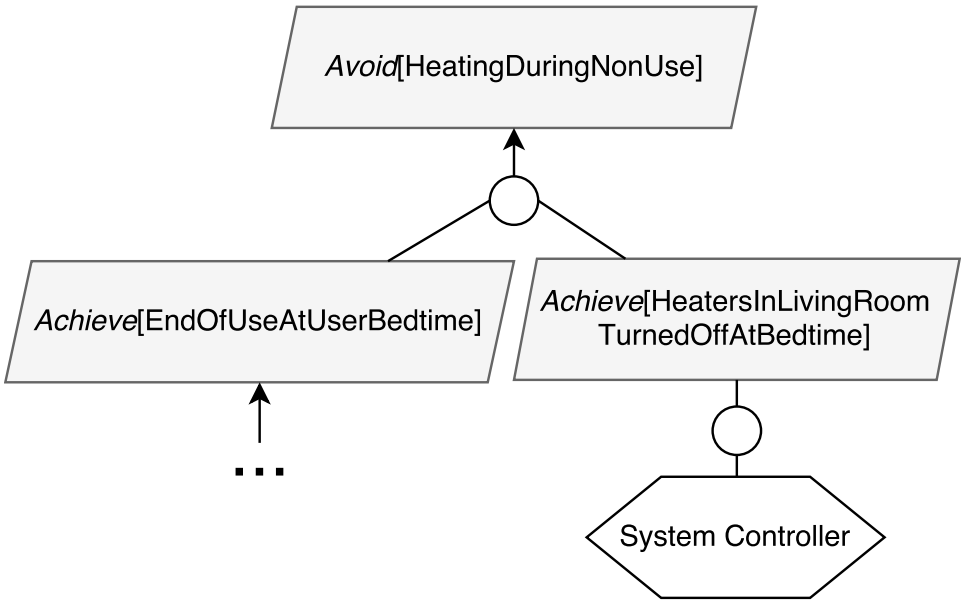

Abbildung 3.4: Goal Diagram - Ziel für das Vermeiden von Heizen ausserhalb der Nutzzeit

3.3.2 Erstellen des Goal Diagrams

Um ein vollständiges Goal Diagram zu erarbeiten, empfiehlt es sich, zuerst die offensichtlichen Ziele des System-to-be zu notieren. In den nachfolgenden Schritten können dann die Beziehungen zwischen den Zielen sowie weitere Verfeinerungen und überliegende Ziele gefunden werden.

Für unser Home Automation Beispiel finden wir unsere ersten Ziele in den Beschreibungen aus der Elicitation Phase. Formulierungen wie *"um X zu tun, soll Y getan werden"*, oder *"das System soll X leisten"*, enthalten häufig wichtige Informationen über die Ziele des Systems. Zwei der wichtigsten Ziele werden direkt zu Beginn erwähnt:

> Das System soll dafür sorgen, dass der Wohnbereich des Kunden zu Nutzzeiten stets ein gesundes Wohnklima aufweist. Sowohl in-

nerhalb als auch außerhalb der Nutzzeiten soll der Energiebedarf der Klimatisierung auf ein Minimum reduziert werden.

Der erste Satz beschreibt das Ziel des Systems, für ein gesundes Wohnklima im Wohnbereich zu sorgen, solange dieser genutzt wird. Die Formulierung "stets" deutet stark auf ein *Maintain* Ziel, also ein Ziel was zu jeder Zeit erfüllt sein soll, hin. Wir definieren unser erstes Ziel:

Maintain[HealthyClimateInLivingSpaceDuringUse]

Der zweite Satz beschreibt ein Soft Goal, nämlich die Reduzierung des Energiebedarfs auf ein Minimum. Damit ergibt sich unser zweites Ziel:

Reduce[EnergyConsumption]

Aus der Beschreibung lassen sich noch weitere Ziele ableiten. Ein vorläufiges Goal Diagram mit (noch) unzusammenhängenden Zielen könnte zum Beispiel wie in Abbildung 3.5 aussehen.

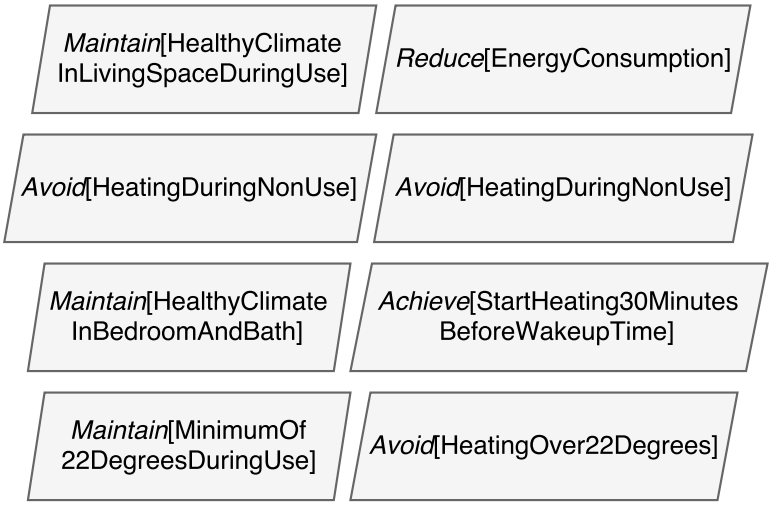

Abbildung 3.5: Goal Diagram - Unzusammenhängendes Goal Diagram, abgeleitet aus der Beschreibung der Elicitation Phase

Der nächste Schritt besteht darin, die vorläufig erarbeiteten Ziele zu verfeinern und überliegende Ziele zu identifizieren. Dies wird so lange gemacht, bis sich ein hierarchisch aufgebautes Goal Diagram ergibt, in dem alle Leaf-Goals einem einzelnen Agenten innerhalb des Systems zugewiesen werden können.

Um überliegende Ziele zu identifizieren und bestehende Ziele, die noch nicht einem einzelnen Agenten zugewiesen werden können, zu verfeinern, stellen wir über ein bereits existierendes Ziel die Fragen **WARUM?** und **WIE?**.

Die Antwort auf die Frage **WARUM?** liefert dabei überliegende Ziele, das heißt Ziele, deren Erfüllung von der Erfüllung des befragten Zieles abhängen. Die Antwort auf die Frage **WIE?** liefert hingegen verfeinerte Ziele, das heißt Ziele, die erfüllt sein müssen, damit auch das befragte Ziel erfüllt ist.

Diese Fragen werden nun für alle Ziele des Goal Diagrams so lange rekursiv gestellt, bis jedes Ziel so weit verfeinert wurde, dass sich dessen Leaf-Goals einem einzelnen Agenten zuordnen lassen und jedes Top-Level Goal ein überliegendes Ziel des Systems beschreibt, welches nicht weiter abstrahiert werden kann.

Nehmen wir zum Beispiel das Ziel des Home Automation Systems *Maintain*[HealthyClimateInLivingSpaceDuringUse]. Die Frage **WARUM?** lässt sich hier nur mit Zielen rechtfertigen, die nicht mehr Teil der Systembeschreibung aus der Elicitation Phase sind (zum Beispiel eine längere Lebensdauer der Nutzer). Das deutet darauf hin, dass dieses Ziel ein Top-Level Goal unseres Systems ist. Die Frage **WIE?** hingegen lässt sich leichter beantworten. Zum einen muss zur Erfüllung des Ziels der Wohnraum während der Nutzzeit stets 22°C oder mehr aufweisen. Zum anderen kann das System das Ziel nur dann erfüllen, wenn wir davon ausgehen können, dass der Wohnraum des Nutzers bereits bei der Installation des Klimatisierungssystems festgelegt wurde.

Diese Erkenntnisse können als ein Ziel und eine Domänenannahme formuliert werden. Abbildung 3.6 zeigt das resultierende Teilmodell.

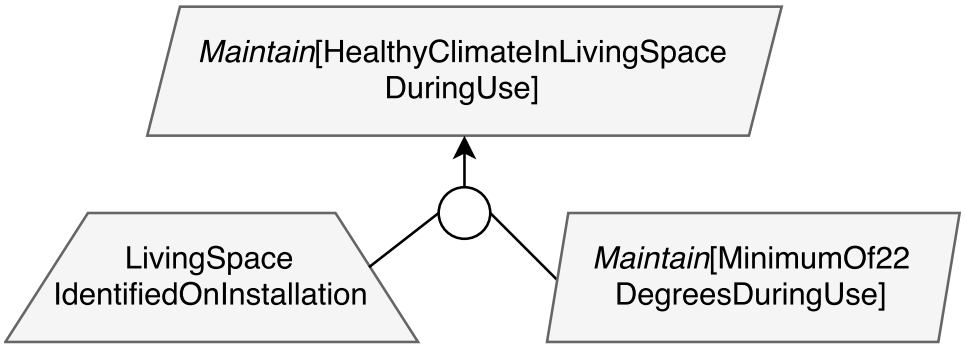

Abbildung 3.6: Goal Diagram - Refinement Goals des Ziels *Maintain*[HealthyClimateInLivingSpaceDuringUse]

Blicken wir zurück auf das unzusammenhängende Goal Diagram aus Abbildung 3.5, stellen wir fest, dass wir eine Verbindung zwischen zwei der abgeleiteten Ziele hergestellt haben, indem wir die **WIE?** Frage für eines der Ziele beantwortet haben. Zusätzlich haben wir eine weitere bisher noch nicht erhobene Assumption identifiziert. Es ist ebenfalls möglich, die Hierarchie zwischen den Zielen durch die Beantwortung der **WARUM?** Frage zu erkennen.

Nehmen wir zum Beispiel das Ziel *Achieve*[StartHeating30MinutesBefore-WakeupTime]. Beantworten wir die Frage, **WARUM?** unser System dieses Ziel erfüllen soll, würden wir herausfinden, dass das System 30 Minuten vor Beginn der Nutzzeit anfangen muss zu heizen, um zu garantieren, dass die Wohnraumtemperatur bereits zu Beginn der Nutzzeit 22°C oder mehr beträgt. Das Ziel *Maintain*[MinimumOf22DegreesDuringUse] beschreibt genau dieses Ziel und kann somit als überliegendes Ziel unseres betrachteten Ziels im Diagramm vermerkt werden.

Betrachten wir nun das Ziel *Maintain*[MinimumOf22DegreesDuringUse], sollte uns auffallen, dass ein 30-minütiges Vorheizen alleine nicht ausreichend ist, um das Ziel zu erfüllen. Das Stellen der **WIE?** Frage beantwortet uns, welche Teilziele wir zusätzlich zu dem bereits identifizierten Ziel noch erfüllen müssen. Neben dem Vorheizen müssen wir zusätzlich garantieren, dass die Weckzeit bereits vom Benutzer eingestellt wurde (damit das System weiß, wann es beginnen muss vorzuheizen) und dass alle Räume mit einem aktiven Heizelement bis auf 22°C aufgeheizt werden.

Die Wahl, dass nur Räume mit aktiven Heizelementen beheizt werden sollen, ist eine Design-Entscheidung, die in einem richtigen Projekt zusammen mit dem Kunden getroffen werden sollte. Für eine Modellierung im Vorfeld könnte zum Beispiel ein generisches Ziel *Maintain*[22DegreesInLivingRoom] definiert werden. Unterschiedliche Lösungsalternativen für dieses Ziel würden als OR Verknüpfungen dargestellt werden. In diesem Fall ist eine Absprache mit dem Kunden von Nöten, um eine korrekte Entscheidung für diesen Teil des Systems zu treffen. Abbildung 3.7 zeigt das aktualisierte Goal Diagram.

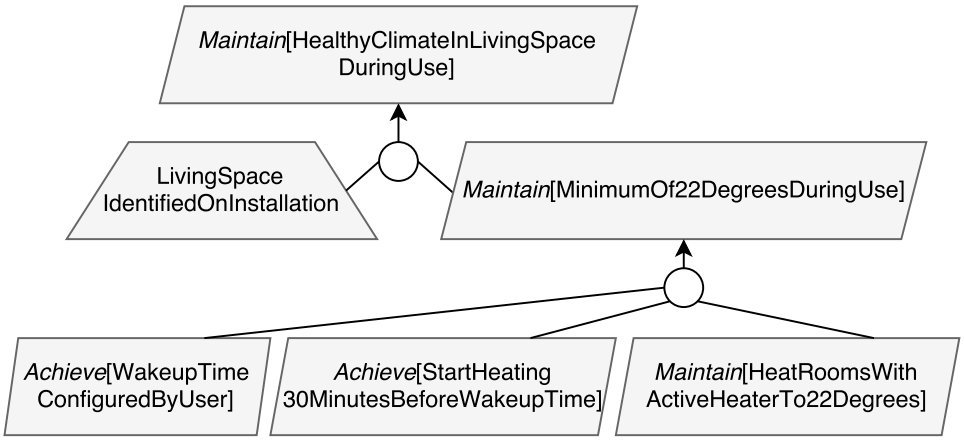

Abbildung 3.7: Goal Diagram - Verfeinerungen des Ziels *Maintain*[MinimumOf22DegreesDuringUse]

Während wir die Ziele weiter verfeinern, lohnt es sich, direkt zu überprüfen, ob die Erfüllung eines neu gefundenen Leaf-Goals bereits

einem einzelnen Agenten zugewiesen werden kann. Lässt sich ein Agent finden, ist das betrachtete Ziel weit genug verfeinert. Das Ziel *Achieve*[WakeupTimeConfiguredByUser] kann zum Beispiel durch den *User* Agenten erfüllt werden, da dieser das zugewiesene Ziel alleine erfüllen kann. Abbildung 3.8 zeigt die Zuweisung des Agenten zu dem Ziel.

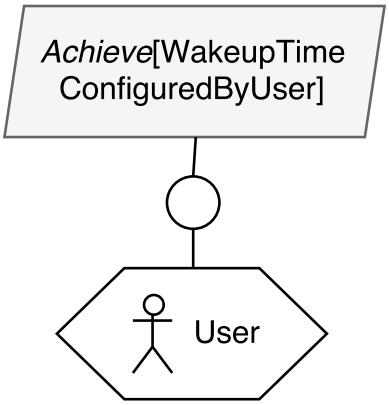

Abbildung 3.8: Goal Diagram - Zuweisung des *User* Agenten zu dem Ziel *Achieve*[WakeupTimeConfiguredByUser]

Um ein vollständiges Goal Diagram zu erhalten, müssen nun die vorangegangenen Schritte rekursiv für jedes Ziel (und alle neu gefundenen) durchgeführt werden. Dieser Prozess ist selbst für ein so simples System wie das Home Automation Beispiel recht zeitintensiv und bedarf meist mehrerer Iterationen, bis ein vollständiges und korrektes Modell erarbeitet wurde. Es ist ebenfalls möglich, dass während des Erstellens der nachfolgenden Modelle Inkonsistenzen oder mögliche Verbesserungen an dem Goal Diagram gefunden werden. In diesem Fall lohnt es sich, das Goal Diagram noch einmal zu überarbeiten und die abgeleiteten Modelle auf ihre Korrektheit zu überprüfen.

Das vollständige Goal Diagram für das Home Automation System kann in Anhang A gefunden werden.

In den nächsten Kapiteln werden wir nun aus dem entstandenen Goal Model das Object Model, das Responsibility Model sowie das Operation Model ableiten.

3.4 Object Model

Das Object Model präsentiert eine strukturelle Sicht auf die konzeptuellen Objekte des Systems. Grafisch wird das Object Model in einem UML-Klassendiagramm-ähnlichen Diagramm dargestellt. Das Object Model wird direkt aus dem Goal Model abgeleitet.

3.4.1 Konzeptuelle Objekte

Ein konzeptuelles Objekt beschreibt eine diskrete Menge von Instanzen eines domänenspezifischen Konzepts. Diese Instanzen haben eine eindeutige Identität, ihre Anzahl kann in jedem Systemzustand festgestellt werden, sie besitzen die gleichen Eigenschaften, unterscheiden sich jedoch in ihrem konkreten Zustand.

Konzeptuelle Objekte ähneln stark den Klassen in objektorientierten Programmiersprachen. Sie beschreiben alle wichtigen Systemteilnehmer mit ihren Eigenschaften. Die tatsächlichen Instanzen der Klasse besitzen dann eine eigene Identität und einen eigenen Zustand, der sie von anderen Instanzen der selben Klasse unterscheidet.

3.4.2 Typen von Konzeptuellen Objekten

Konzeptuelle Objekte werden in die Kategorien *Entity*, *Association*, *Agent* und *Event* unterschieden.

Entities sind autonome und passive Objekte innerhalb des Systems. Sie können im Gegensatz zu Assoziationen unabhängig von anderen Objekten im System existieren. Im Gegensatz zu Agenten können Entitäten nicht das Verhalten von anderen Objekten im System kontrollieren. In dem Home Automation System würden die konzeptuellen Objekte *Heater* und *Room* als Entity modelliert werden.

Agenten sind autonome und aktive Objekte. Agenten nehmen aktiv Einfluss auf den Systemverlauf, indem sie das Verhalten anderer Objekte kontrollieren. Die Entscheidungen eines Agenten werden durch den Zustand des Agenten selbst sowie die Zustände und Aktionen anderer Systemteilnehmer beeinflusst. In dem Home Automation System sind der *User*, der *System Controller* sowie alle Sensoren konzeptuelle Objekte, die eine Agentenrolle füllen.

Assoziationen sind konzeptuelle Objekte, die andere konzeptuelle Objekte innerhalb des Systems verknüpfen. Jedes verknüpfte Element spielt eine bestimmte Rolle in der Assoziation. Ein vergleichbares Konzept aus der Informatik ist eine Datenbank-Relation im ER-Modell. In dem Home Automation System können wir die Tatsache, dass jeder Raum null bis ein Heizelement besitzt als eine binäre Assoziation modellieren.

Ein **Event** ist ein spontanes Ereignis innerhalb des Systems. Das bedeutet, dass ein Event nur in einem einzigen bestimmten Systemzustand existieren kann. In dem Home Automation System lassen sich zum Beispiel die Events *30MinutesBeforeWakeupTime* und *Bedtime* finden.

3.4.3 Grafische Darstellung

Das Object Model wird in einem UML-Klassendiagramm-ähnlichen Diagramm dargestellt.

Entitäten und Agenten werden als UML-Klassen ohne Operationen dargestellt. Die Operationen der Agenten werden später in dem Operation Model modelliert. Attribute von Entities und Agenten werden, optional mit Typ des Attributs (String, Integer, usw), in einem Kästchen unter dem Klassennamen festgehalten. Vererbung wird wie im UML Klassendiagramm auch durch Pfeile zwischen den Objekten dargestellt. Wie in dem Goal Diagram, können zusätzliche Informationen, wie textuelle Beschreibungen, als Annotationen innerhalb des Diagramms oder in einem separaten Dokument hinterlegt werden. Eine textuelle Beschreibung der konzeptuellen Objekte in einem separaten Dokument bietet sich besonders für komplexe Systeme an, da diese Beschreibungen im weiteren Projektverlauf als Definitions-Glossar genutzt werden können, so dass alle an dem Projekt beteiligten Parteien wissen, was mit einem bestimmten Begriff innerhalb der Domäne gemeint ist. Kapitel 5.1 geht näher auf die Vorteile eines eindeutigen Vokabulars ein. In dem Definitions-Glossar würde für das Home Automation System zum Beispiel stehen, dass mit dem Begriff *User* ältere Menschen bezeichnet werden, an die sich das System richtet.

Abbildung 3.9 zeigt die Entität *Bedroom*, sowie die Agenten *Movement-Sensor* und *Alarm*.

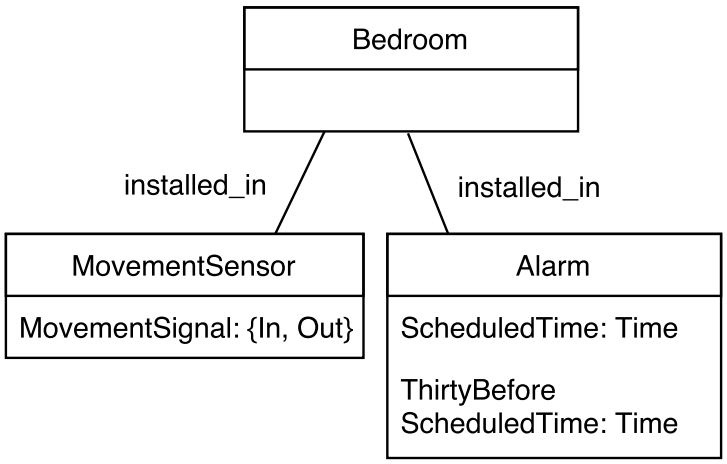

Abbildung 3.9: Object Diagram - Ausschnitt der Konzepte *Bedroom*, *MovementSensor* und *Alarm*

Assoziationen werden als UML-Assoziationen (Kante zwischen Objekten) im Fall von binären Assoziationen und als Relationsknoten (Diamant wie im ER-Modell) im Fall von n-ären Assoziationen dargestellt. Assoziationen können benannt werden und jeder Teilnehmer der Assoziation bekommt eine Arität zugewiesen. Sollten bei einer Assoziation einer der Teilnehmer genau ein mal an der Assoziation teilnehmen, kann die Arität weggelassen werden. Sollte die Assoziation eigene Attribute besitzen, werden diese als

separate UML-Klasse modelliert und mit einer gestrichelten Linie mit der Assoziation verbunden. Abbildung 3.9 zeigt das Schlafzimmer, in welchem genau einen Bewegungssensor und ein Wecksystem installiert sind.

Events werden nur in dem Behaviour Model benötigt. Da in dieser Arbeit kein Behaviour Model erstellt wird, werden wir uns nicht weiter mit Events beschäftigen.

3.4.4 Erstellen des Object Models

Wie eingangs erwähnt, wird das Object Model direkt aus dem Goal Model abgeleitet. Dafür werden die Ziele auf konzeptuelle Objekte und Assoziationen zwischen diesen Objekten untersucht. Sollten während der Erstellung des Goal Models Annotationen für die Ziele erstellt worden sein – das heißt zusätzliche Informationen, textuelle Beschreibungen, usw – sind diese ebenfalls wertvolle Quellen, um Informationen für das Erstellen des Object Models zu erhalten.

Schauen wir uns zum Beispiel das Ziel *Maintain*[HealthyClimateInLivingSpaceDuringUse] an. In diesem Ziel tauchen die potentiellen Konzepte *Climate* und *LivingRoom* auf. *Climate* ist kein guter Kandidat für ein konzeptuelles Objekt, da sich der Begriff allgemein auf das Klima des gesamten Wohnbereichs bezieht. *LivingSpace* hingegen ist ein guter Kandidat für ein konzeptuelles Objekt, da das System die Heizungen des Wohnraums kollektiv an- und abschaltet.

Auf die selbe Art und Weise arbeiten wir uns nun durch die Ziele des Goal Models und suchen nach weiteren konzeptuellen Objekten. Manche konzeptuellen Objekte können erst durch das Betrachten mehrerer Ziele vollständig modelliert werden. Für unser Home Automation System würden wir im weiteren Verlauf des Modellierens zum Beispiel herausfinden, dass sich eine Wohnung in mehrere Räume aufteilen lässt. Das Schlafzimmer und das Badezimmer können wir dann durch Vererbung ebenfalls als Zimmer modellieren. Der Wohnraum hingegen ist an sich kein Raum, sondern beschreibt eine Menge von Räumen. Im Gespräch mit dem Kunden könnte dann geklärt werden, wie die Räume einer Wohnung bei der Installation in Schlafzimmer, Badezimmer und Wohnraum aufgeteilt werden. Eine Möglichkeit dieses Problem zu lösen, wäre zum Beispiel, dass die Ingenieure bei der Installation jedem Raum eine eindeutige Identifikationsnummer zuweisen und diese dazu nutzen, um in dem System eine Aufteilung vorzunehmen. Das wäre ein Konzept, dass in dem Goal Model nicht abgebildet wird, aber für die Vollständigkeit des Object Models äußerst wichtig ist. Diese Identifikationsnummer könnten wir dann der *Room* Entität als Attribut hinzufügen.

Abbildung 3.10 zeigt die konzeptuellen Objekte *Bedroom* und *Bath*, welche von dem Konzept *Room* erben. Der Wohnraum besteht aus beliebig vielen Räumen.

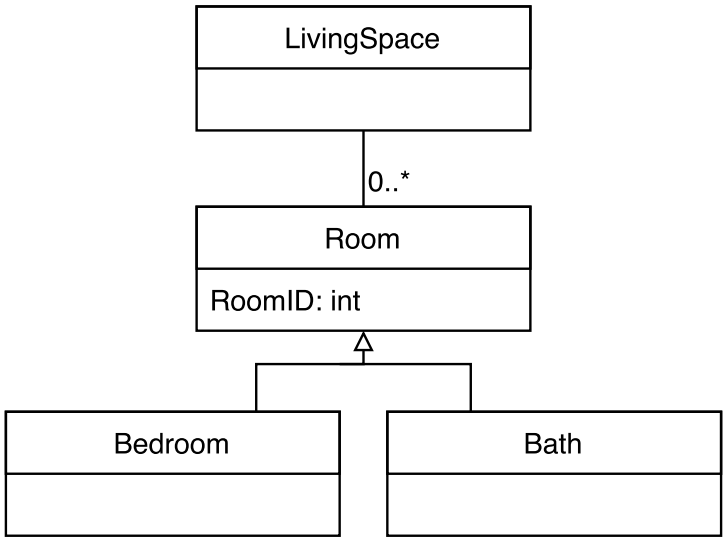

Abbildung 3.10: Object Diagram - Zusammenhang der Konzepte *Room*, *Bedroom*, *Bath* und *LivingSpace*

Wichtig bei der Erstellung des Object Models ist es, die noch zu erstellende Kern-Software, wie den System Controller, nicht mit zu modellieren. Es wird im Object Model davon ausgegangen, dass die Software vollen Zugriff auf die Instanzen der unterschiedlichen Konzepte zur Laufzeit hat. Was jedoch modelliert werden sollte sind Komponenten, die innerhalb der Domäne als eigenes Konzept verstanden werden können. Ein solches Konzept wäre zum Beispiel der *Timer*, da dieser in der Domäne als eigenständiges konzeptuelles Objekt gehandhabt wird. Ob die Komponente schlussendlich Teil der zu erstellenden Software ist oder eine separate Komponente (Hardware, Drittanbieter-Software), ist implementationsabhängig.

Ein vollständiges Object Model des Home Automation System kann in Anhang A gefunden werden. Einige der Konzepte, wie die *HeatingFormula* stehen in dem Modell alleine, was jedoch nur ausdrückt, dass dieses Konzept in keiner direkten Relation zu einem der anderen Konzepte steht. Für die korrekte Funktionalität des Systems sind diese Konzepte jedoch unabdingbar.

3.5 Responsibility Model

Nachdem wir ein vollständiges Object Model erstellt haben, wird es Zeit, dass wir uns Gedanken über die Verantwortlichkeiten innerhalb des Systems machen. Um dies zu erreichen, erstellen wir das Responsibility Model, welches die Verantwortlichkeiten der Agenten innerhalb des Systems aufzeigt und welche Daten die Agenten untereinander austauschen. Grafisch

präsentieren wir das Responsibility Model in einem *Agent Diagram*. Das Agent Diagram zeigt, welche Ziele ein Agent innerhalb des Systems erfüllt sowie die Variablen, die ein Agent beobachtet und/oder kontrolliert.

3.5.1 Agenten und Agententypen

Ein Agent ist wie bereits erwähnt ein konzeptuelles Objekt, das aktiv Einfluss auf den Systemverlauf hat, indem es das Verhalten von anderen Objekten kontrolliert. Agenten sind dementsprechend die einzigen Systemteilenehmer, die aktiv zu der Erfüllung eines Systemziels beitragen können. Um dies zu tun, müssen Agenten den Systemzustand beobachten und sorgfältig entscheiden, wie sie die anderen Systemteilnehmer manipulieren, um die ihnen zugewiesenen Systemziele zu erfüllen. Agenten werden in vier Kategorien unterteilt.

Neue Software Agenten sind Softwarekomponenten, die noch entwickelt werden müssen. Dazu gehören zum Beispiel Software Controller und Web- oder Microservices.

Existierende Software Agenten sind Softwarekomponenten, die bereits entwickelt wurden. Dazu gehören interne Legacy Software, eingekaufte Software und fremde Komponenten wie öffentliche APIs.

Geräte sind reale Hardware, wie Sensoren, Aktoren, Messinstrumente und Kommunikationsgeräte. Geräte sind häufig in Systemen anzutreffen, in denen die Benutzerinteraktion nicht über ein bereits vorhandenes Interaktionsmedium (Computer, Mobiltelefon) geschieht.

Human Agents sind Benutzer und Fachleute, die mit dem System interagieren und einen Teil zu dem korrekten Systemablauf beitragen. In dem Home Automation System hat der Benutzer zum Beispiel die Aufgabe, die Weckzeit im System einzustellen.

Außer den neuen Software Agenten sind alle anderen Agenten Environment Agenten. Das heißt, dass das System diese Agenten nicht kontrollieren kann. Auch existierende Software Agenten zählen zu den Environment Agenten und nicht zu den Software Agenten, da selbst interne Legacy Software nicht direkt Teil der neu entwickelten Systemsoftware ist.

3.5.2 Agent Diagram

Das Agent Diagram ist eine grafische Repräsentation, welche die verschiedenen Systemagenten und ihre Verantwortlichkeiten darstellt. Das Agent Diagram wird als annotierter Graph dargestellt, in dem die Agenten des Systems in der Mitte als Knoten dargestellt sind. Auf der linken und der rechten Seite der Agenten werden unterschiedliche Systeminformationen präsentiert.

Zur linken Seite der Agenten sind die Ziele dargestellt, welche der Agent in dem System zu erfüllen hat. Die Ziele werden mit

den Agenten genau so verbunden, wie aus dem Goal Diagram bekannt. Eine solche Verbindung zwischen Agent und Ziel wird *Responsibility Link* genannt. In Abbildung 3.11 ist der Environment Agent *Temperature Sensor* zu sehen, der für die Erfüllung des Zieles *Maintain*[TemperatureSensorDeliversAccurateMeasurementsForRoom] verantwortlich ist.

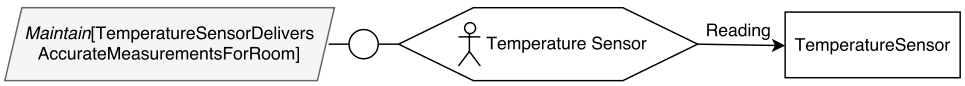

Abbildung 3.11: Agent Diagram - Teildiagramm für den Agenten *Temperature Sensor*

Zur rechten Seite der Agenten werden die konzeptuellen Objekte des Systems dargestellt und mit den Agenten verknüpft. Eine eingehende Pfeilkante von einem Objekt zu einem Agenten bedeutet, dass der Agent die Attribute dieses Objektes beobachtet. Eine ausgehende Pfeilkante von einem Agenten zu einem Objekt bedeutet, dass der Agent die Attribute des Objektes kontrolliert. Welche Attribute beobachtet oder kontrolliert werden, wird durch eine Beschriftung der Kanten mit den entsprechenden Attributen angezeigt. Ist eine Kante nicht beschriftet, beobachtet oder kontrolliert der Agent alle Attribute des Objektes. Eine solche Verbindung zwischen Agent und Objekt wird *Monitoring Link* für beobachtende Aktionen und *Control Link* für kontrollierende Aktionen genannt. In Abbildung 3.11 ist der Agent *Temperature Sensor* zu sehen, der das Attribut *Reading* des konzeptuellen Objektes *TemperatureSensor* kontrolliert. Diese Verknüpfung von gleichnamigen Agenten und konzeptuellen Objekten ist ein wiederkehrendes Muster, um die realen Geräte-Agenten mit ihrer Repräsentation in dem Object Model zu verknüpfen.

Genau wie bei allen vorherigen Modellen können zusätzliche Informationen, wie eine textuelle Beschreibung oder der Typ eines Agenten direkt im Agent Diagram, oder in einem separaten Dokument als Annotation hinterlegt werden.

3.5.3 Erstellen des Agent Diagrams

Der erste Schritt bei der Erstellung des Agent Diagrams wird uns sehr leicht gemacht, da wir bereits beim Erstellen des Goal Diagrams die Verantwortlichkeiten für die Erfüllung der Leaf-Goals identifiziert haben. Das bedeutet, dass wir ledeglich alle Ziele, für die ein Agent verantwortlich ist, extrahieren und in dem Agent Diagram verknüpfen müssen. Abbildung 3.12 zeigt die Verantwortlichkeiten des Agenten *HeatingFormula*.

Im nächsten Schritt geht es darum herauszufinden, welche Attribute die Agenten in dem System beobachten und welche sie kontrollieren. Bei

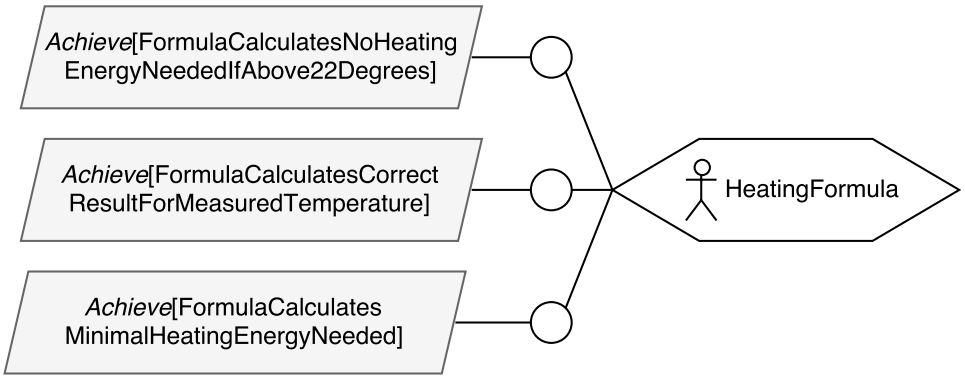

Abbildung 3.12: Agent Diagram - Verantwortlichkeiten des Agenten *HeatingFormula*

diesem Schritt hilft es, sich Goal Model und Object Model anzuschauen und herauszufinden, welche Informationen ein Agent für die Erfüllung seiner Ziele benötigt (zu beobachtende Attribute) und welche Attribute kontrolliert werden müssen, um die zugewiesenen Ziele zu erfüllen (zu kontrollierende Attribute).

Schauen wir uns zum Beispiel den Agenten *User* an. Das einzige Ziel, wofür der Benutzer verantwortlich ist, ist das Einstellen der Weckzeit.

Das Einstellen der Weckzeit ist eine Aktion, mit der der Benutzer ein Attribut des konzeptuellen Objektes *Alarm* manipuliert. Wir halten dies in unserem Diagramm fest, indem wir eine ausgehende Pfeilkante von dem *User* Agenten zu dem *Alarm* Objekt zeichnen und die Kante mit dem kontrollierten Attribut *ScheduledTime* beschriften. Abbildung 3.13 zeigt das resultierende Agent Diagram.

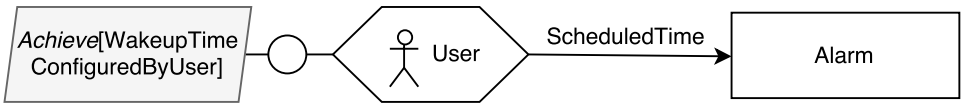

Abbildung 3.13: Agent Diagram - Verantwortlichkeiten und kontrollierte Attribute des *User* Agenten

Komplizierter wird es, wenn für die Erfüllung eines Ziels ebenfalls Informationen benötigt werden. Um die benötigte Heizleistung zu errechnen, braucht der Agent *HeatingFormula* eine aktuelle Raumtemperatur. Die Heizformel-Systemkomponente muss dementsprechend Informationen über die aktuelle Temperatur von der Systemsoftware erhalten, um anschließend die errechnete Heizleistung an das System zurück zu schicken. Das geschieht, indem der Agent *HeatingFormula* die Informationen seines eigenen konzeptuellen Objektes liest und schreibt. Das konzeptuelle Objekt *HeatingFormula* stellt diese Informationen dann dem *System Controller* Agenten bereit.

Dieses Durchschleusen von den Informationen eines Environment Agenten zu einem Software Agenten (und umgekehrt) über ein entsprechendes konzeptuelles Objekt ist ein wiederkehrendes Muster bei der Erstellung des Agent Diagrams. Abbildung 3.14 zeigt das resultierende Agent Diagram des Agenten *HeatingFormula*.

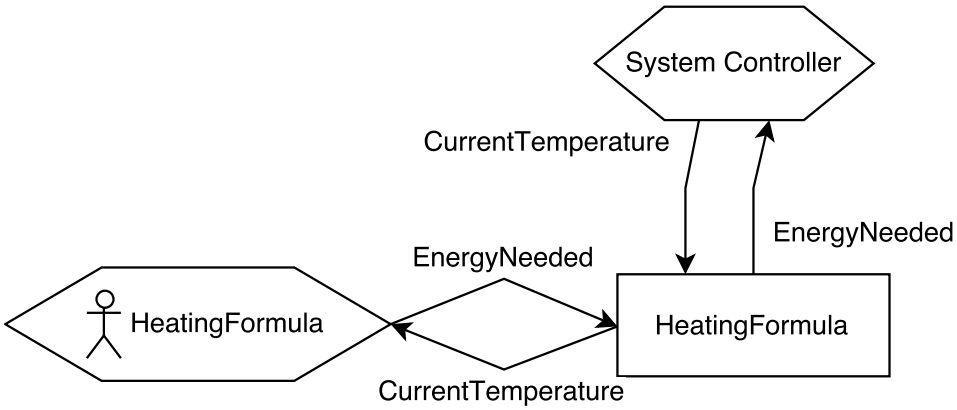

Abbildung 3.14: Agent Diagram - Informationsaustausch zwischen den Agenten *Heating Formula* und *System Controller*

So verfahren wir nun für alle weiteren Agenten, die wir in dem Goal Diagram identifiziert haben. Das vollständige Agent Diagram kann in Anhang A gefunden werden. In dem resultierenden Agent Diagram spielt der *System Controller* eine zentrale Rolle. Das ist nicht ungewöhnlich, da der *System Controller* in dem Home Automation System primär als Mediator zwischen den Sensoren und der Heizanlage für die unterschiedlichen Räume agiert und zusätzlich noch die Steuerung des gesamten Heizsystems bei Nutz- und Nichtnutzzeiten übernimmt.

3.6 Operation Model

Das letzte Modell, das im Rahmen dieser Arbeit erstellt werden soll, ist das Operation Model. Das Operation Model bietet eine funktionale Sicht auf das System, indem es zu allen Leaf-Goals aus dem Goal Diagram die Operationen auflistet, die erforderlich sind, um das entsprechende Ziel zu erfüllen. Das Operation Model wird grafisch in dem sogenannten *Operationalization Diagram* dargestellt.

3.6.1 Operationen und Operationstypen

Eine Operation ist eine Aktion eines Agenten, die den Systemzustand ändert. Inputs von Operationen sind konkrete Attributwerte der Instanzen

von konzeptuellen Objekten, die beeinflussen, wie genau die Operation durchgeführt wird. Outputs von Operationen sind konkrete Attributwerte der Instanzen von konzeptuellen Objekten, die manipuliert werden, um das System in einen gewünschten Zustand zu bringen. Operationen werden in zwei Kategorien unterteilt.

Software Operationen werden von Software Agenten durchgeführt. Häufig automatisieren sie Operationen, die in dem System-as-is manuell durchgeführt werden müssen. Solche Operationen werden auch *Services* oder *Functional Features* genannt.

Environment Operationen werden von menschlichen Agenten, Geräten oder existierender Software durchgeführt. Auf diese Operationen hat die Software keinen Einfluss.

In dem Home Automation System wäre zum Beispiel das Starten des *Timers* eine Software Operation, da der *System Controller* bei Empfang einer Bewegungsmeldung den *Timer* für das Erkennen der Bettruhe startet. Das Einstellen der Weckzeit hingegen ist eine Environment Operation, da der *User* dafür verantwortlich ist, die Weckzeit am Alarm einzustellen. Der *System Controller* hat hier keinen Einfluss darauf, wann und wie die Weckzeit eingestellt wird. Um die neue Weckzeit zu erhalten, muss der *System Controller* dementsprechend das Attribut *ScheduledTime* des *Alarm* Objektes beobachten.

3.6.2 Operationalization Diagram

Das Operationalization Diagram ordnet allen Leaf-Goals aus dem Goal Model eine entsprechende Operation zu, die der verantwortliche Agent durchführen muss, um das Ziel zu erfüllen.

Die Operationen werden den Zielen mit der selben Notation zugewiesen, wie auch Agenten. Der Operationsname sollte aussagekräftig sein, sodass das Operationalization Model einfach verständlich ist. Grafisch werden Operationen als Ellipse dargestellt. Abbildung 3.15 zeigt das Ziel *Achieve*[StartTimer], welches durch die Operation *StartTimer* realisiert wird.

Jede Operation kann Inputs und Outputs besitzen, die Attribute von Objekten innerhalb des Systems lesen oder setzen. Ausgehende Pfeilkanten von einer Operation zu einem konzeptuellen Objekt stellt dabei einen Output, das heißt eine Änderung eines Attributes dar. Eine eingehende Pfeilkante von einem konzeptuellen Objekt zu einer Operation hingegen stellt einen Input dar, das heißt einen Attributwert, den die Operation benötigt, um einen korrekten Output zu generieren. Welche Attribute bei den Input- und Output-Verbindungen gelesen oder gesetzt werden, wird wie beim Agent Diagram auch an die Kante zwischen Operation und Objekt geschrieben. Es ist außerdem auch möglich, dass Operationen nur intern (in dem konzeptuellen Objekt des Agenten) Attribute lesen und/oder verändern. Abbildung 3.15 zeigt die Operation *StartTimer*, die als Output

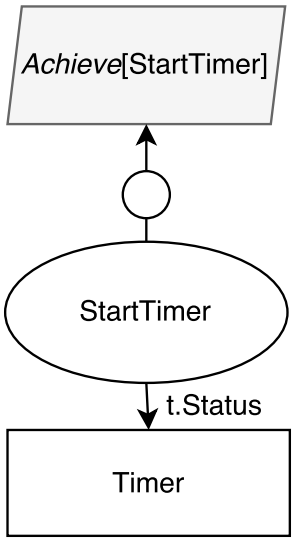

Abbildung 3.15: Operationalization Diagram - Starten des Timers als Operation des System Controllers. "t" ist ein symbolischer Name, um gleichnamige Attribute von unterschiedlichen Objekten zu kennzeichnen.

das Attribut *Status* des Timers setzt.

Wie bei allen anderen Modellen auch, lohnt es sich, zusätzliche Informationen wie Pre- und Post-Conditions sowie eine textuelle Beschreibung als Annotation direkt in dem Diagramm oder in einem separaten Dokument zu hinterlegen.

3.6.3 Erstellen des Operationalization Diagrams

Beim Erstellen des Operationalization Diagrams bietet es sich an, ein kleineres Operationalization Diagram für jeden Agenten anzufertigen. So ist es einfacher, die Operationen eines Agenten zu identifizieren und nachzuvollziehen, welche Ziele mit Hilfe der Operation erfüllt werden.

Der erste Schritt besteht darin, wie beim Agent Model auch, alle Leaf-Goals im Goal Diagram zu finden, die der betrachtete Agent erfüllen muss. Sind alle Leaf-Goals gefunden, müssen wir uns überlegen, welche Operationen der betrachtete Agent durchführen muss, um das gewünschte Ziel zu erreichen.

Nehmen wir zum Beispiel den *System Controller* Agenten. Dieser ist dafür verantwortlich, das Ziel *Achieve*[StartLivingSpaceHeaters-AtScheduledTime] zu erfüllen.

Zur Erfüllung dieses Ziels benötigt der *System Controller* die Uhrzeit, welche 30 Minuten vor der Weckzeit liegt. In dem Agent Diagram haben wir bereits festgehalten, welche Attribute anderer Agenten der *System Controller* beobachtet. Die Uhrzeit erhält der *System Controller* von dem

Alarm. Um ein Heizelement anzuschalten, muss der *System Controller* das *Status* Attribut eines Heizelementes auf *On* setzen.

Wir modellieren diese Annahmen in dem Operationalization Diagram, indem wir eine neue Operation erstellen, die das Anschalten der Heizelemente zu der gegebenen Uhrzeit übernimmt. Wir nennen diese Operation *StartHeatersAtScheduledTime* und fügen als Input das *ThirtyBeforeScheduledTime* Attribut des *Alarm* Objektes hinzu. Als Output setzt der *System Controller* das *Status* Attribut des *Heater* Objektes. Um etwaige gleichnamige Attribute unterscheiden zu können, beschriften wir die Kanten der Inputs und Outputs zusätzlich zu dem gelesenen oder manipulierten Attribut mit einem symbolischen Namen. Die Verknüpfung des symbolischen Namens und des Attributes erfolgt in programmatischer Punktnotation. Für das Heizelement benutzen wir zum Beispiel den symbolischen Namen *"h"*. Würden wir in der selben Operation als Output noch andere Attribute mit dem Namen *Status* setzen, könnten wir die Zielobjekte anhand ihrer symbolischen Namen auseinanderhalten. Eine gute Benennung ist dabei wichtig, da keine explizite Zuordnung zu den referenzierten Objekten existiert.

Das resultierende Operationalization Diagram ist in Abbildung 3.16 dargestellt.

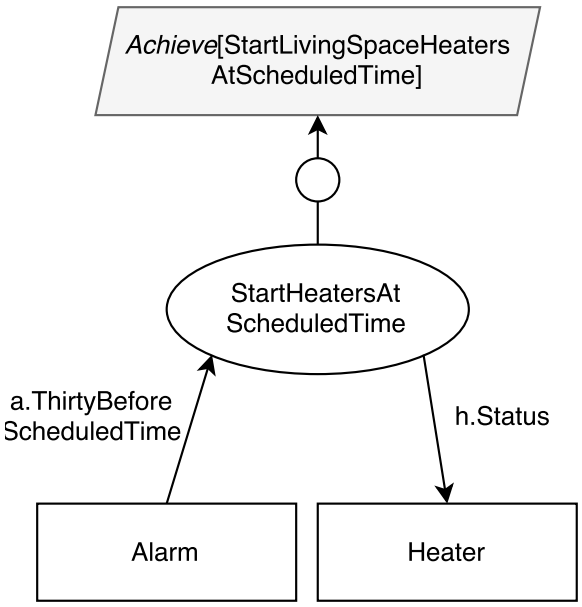

Abbildung 3.16: Operationalization Diagram - Die *RegisterInUse* Operation des *System Controller* Agenten

So verfahren wir nun für alle weiteren Leaf Goals unseres *System Controllers*. Dabei ist es möglich, dass manche Operationen mehr als nur ein Ziel erfüllen oder die Erfüllung eines Zieles mehrere Operationen erfordert.

Die vollständigen Operationalization Diagramme für alle Systemagenten können in Anhang A gefunden werden.

3.7 Zusammenfassung

In diesem Kapiel wurde das KAOS Framework für einen modellorientierten Ansatz im Goal-oriented Requirements Engineering vorgestellt. Anhand eines Fallbeispieles wurde erläutert, wie ein Goal Model in Form eines Goal Diagrams modelliert werden kann und wie das Object Model, das Agent Model und das Responsibility Model aus diesem abgeleitet werden können.

Dieses Kapitel diente dabei als Schnelleinführung in das KAOS Framework. Das Obstacle Model und das Behaviour Model wurden nicht erstellt und auf weiterführende Techniken wurde aus Platzgründen nicht weiter eingegangen. Eine vollständige Abhandlung über das Erstellen eines System Models mit Hilfe von KAOS kann in dem zweiten Teil des Buches *Requirements Engineering - From System Goals to UML Models to Software Specifications* [21] von Professor Axel van Lamsweerde gefunden werden.

Einer der Teilbereiche, den KAOS nicht abdeckt, ist das Erstellen einer tatsächlichen Spezifikation. Im folgenden Kapitel werden wir uns damit beschäftigen, eine szenarionbasierte Spezifikation aus den in diesem Kapitel erstellten Modellen abzuleiten.

Kapitel 4

Szenariobasierte Spezifikation

In diesem Kapitel wird präsentiert, wie eine szenariobasierte Spezifikation systematisch, aus den mit KAOS erarbeiteten Modellen, abgeleitet werden kann. Als Grundlage für die Spezifikation dienen die in Kapitel 3 erarbeiteten KAOS Modelle.

4.1 ScenarioTools – Überblick

ScenarioTools ist eine Sammlung von Eclipse Plugins, welche auf dem Eclipse Modeling Framework aufbauen und es ermöglichen, szenariobasierte Spezifikationen in Eclipse zu erstellen, die Spezifikation zu simulieren und mithilfe der Controller Synthese die Spezifikation automatisiert auf Fehler zu überprüfen.

Eine ScenarioTools Spezifikation setzt sich aus vier Bestandteilen zusammen.

Das **Klassenmodell** beschreibt die Objekte des zu spezfizierenden Systems sowie ihre Attribute und Assoziationen zu anderen Objekten. Das Klassenmodell enthält außerdem einen System Container für das spezifizierte System, in dem alle für die Simulation benötigten Objekte hinterlegt werden. Modelliert wird das Klassenmodell mit dem Eclipse Modeling Framework in Form eines Ecore-Datenmodells.

Das **Objektmodell** beschreibt die instanzierten Objekte des Systems sowie ihre Attribute zur Laufzeit. In ScenarioTools übernimmt eine Dynamische Instanz des System Containers die Rolle des Objektmodells.

Die **SML Spezifikation** bildet das Herzstück eines ScenarioTools Projektes. SML steht für *Scenario Modeling Language* und ist eine domänenspezifische Sprache für das Schreiben von formalen, szenariobasierten Spezifikationen. Die Sprache wurde an dem Fachgebiet für Software Engineering der Leibniz Universität Hannover entwickelt und ist eine textbasierte

Variante der Life Sequence Charts mit erweiterten Funktionen [8].

Die **Runconfiguration** verknüpft die SML Spezifikation mit dem Objektmodell. Die in der Spezifikation beschriebenen Rollen werden konkreten Objekten in der Dynamischen Instanz des System Containers zugewiesen. Das ermöglicht es, verschiedene Datenmodelle und SML Spezifikationen mit Hilfe spezieller Konfigurationen zu kombinieren.

Sind alle Teile einer szenariobasierten Spezfikation erstellt, kann die Spezifikation in ScenarioTools simuliert werden. Bei der **Simulation** entscheidet der Benutzer, welche Nachrichten die Systemobjekte austauschen. Das ermöglicht es dem Benutzer, die Spezifikation per Hand durchzuspielen und etwaige Unstimmigkeiten und Fehler in der Spezifikation aufzudecken [8].

Neben der Simulation gibt es noch die **Controller Synthese**, welche vollautomatisch alle möglichen Systemzustände und Transitionen berechnet und nach Fehlern in der Spezifikation sucht. Gefunden werden *Safety Violations* (Systemzustand verletzt die Spezifikation) und *Liveness Violations* (guter Systemzustand wird nie erreicht) [8].

Im weiteren Verlauf dieses Kapitels werden wir eine szenariobasierte Spezifikation mit ScenarioTools für das Home Automation Fallbeispiel aus Kapitel 3 anfertigen.

4.2 Erstellen des Klassenmodells

Der erste Schritt beim Erstellen einer szenariobasierten Spezifikation mit ScenarioTools ist das Erstellen des Klassenmodells. Alle wichtigen Informationen für das Klassenmodelll finden sich in dem Object Model und dem Responsibility Model wieder.

Das Klassenmodell hat starke Ähnlichkeiten mit einem objektorientierten Domänenmodell. Konzeptuelle Objekte werden als Klassen modelliert, Objektattribute können als Attribute (zB. für primitive Datentypen und Collections) oder als Referenzen (für Assoziationen unter Objekten) modelliert werden. Operationen spielen in ScenarioTools eine wichtige Rolle, da das Austauschen von Informationen unter den Objekten nachrichtenbasiert erfolgt. Da das Eclipse Modeling Framework nativ keine nachrichtenbasierte Kommunikation unterstützt, modellieren wir eine Nachricht als eine Operation bei dem Empfänger.

Alle konzeptuellen Objekte aus dem Object Model sowie ihre Assoziationen können in das Klassenmodell übernommen werden. Jede Klasse sollte dabei von der Klasse *NamedElement* erben, um den Instanzen der Klasse einen Namen für die Simulation und Controller Synthese zuweisen zu können. Binäre Assoziationen können modelliert werden, indem beide Objekte eine Referenz auf den Partner erhalten und diese Referenzen als *EOpposite* voneinander deklariert werden. Zusätzlich müssen wir noch die primäre Software Komponente (den *SystemController*) unseres Systems

und einen System Container erstellen, der alle Objekte unseres Systems beinhaltet. Der System Container ist notwendig, damit wir später die Spezifikation und das Klassenmodell über eine Dynamische Instanz des Containers verknüpfen können.

Das resultierende Klassenmodell ist in Abbildung 4.1 dargestellt. Alle konzeptuellen Objekte sowie ihre Assoziationen wurden aus dem Object Model übernommen. Die *HomeautomationSystem* Klasse dient in unserem Projekt als System Container. Es ist wichtig anzumerken, dass in dem dargestellten Klassenmodell noch keine Attribute und keine Operationen modelliert wurden.

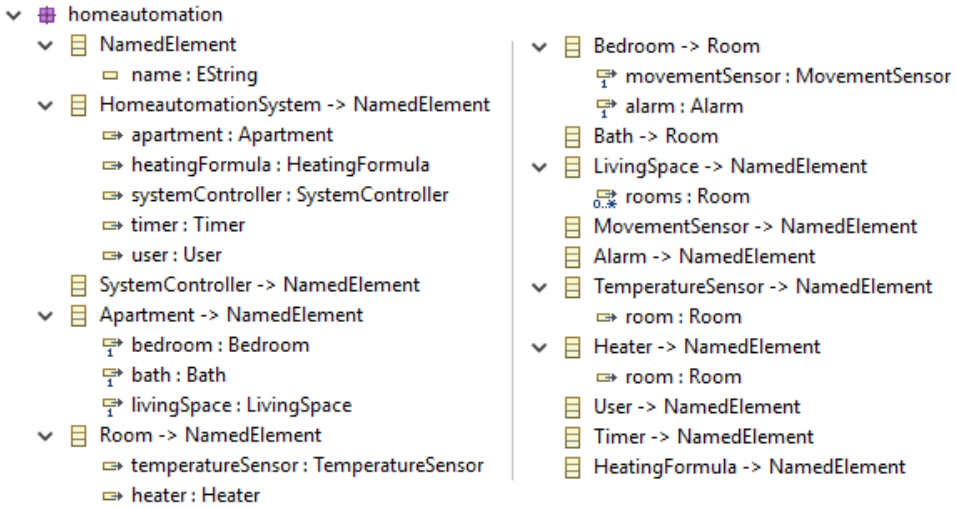

Abbildung 4.1: Klassenmodell - Abgeleitetes Klassenmodell, ohne Attribute und Operationen. Aus Platzgründen wird das Klassenmodell nicht in graphischer Form dargestellt.

Die Attribute der konzeptuellen Objekte können in dem Ecore Datenmodell auf zwei unterschiedliche Weisen modelliert werden. Zum einen kann ein Attribut unverändert als Klassenattribut modelliert werden. Das macht Sinn bei festen Attributen, wie der *RoomId* des *Room* Objektes. Da der Informationsaustausch zwischen den Objekten nachrichtenbasiert erfolgt, kann es zum anderen Sinn machen, ein Attribut als Satz von Attributverändernden Operationen zu modellieren. Nehmen wir zum Beispiel das Attribut *Status* des Objektes *Heater*. Wollen wir erreichen, dass der *System-Controller* das Heizelement ausschaltet, muss dieser eine Nachricht an das Heizelement senden. Modellieren wir *Status* als Enum-Attribut des *Heaters*, müssten wir eine *setStatus(HeaterStatus:Off)* Nachricht senden. Wir können jedoch auch die zwei Operationen *activate()* und *deactivate()* erstellen, die von dem Controller auf dem Zielobjekt aufgerufen werden.

Für das An- und Ausschalten von Systemkomponenten wirken *ac-*

tivate()/deactivate() Operationen natürlicher als Setter-Operationen. Bei anderen Attributen, wie zum Beispiel dem *MovementSignal* des *MovementSensors*, ist eine Operation wie *signalMovement(MovementSignal:In)* jedoch gut verständlich und verringert die Anzahl an benötigten Operationen.

Den letzten Teil des Klassenmodells bilden die Operationen der Objekte. Ein Großteil der Operationen, die dabei modelliert werden, sind Nachrichten für den Austausch von Informationen zwischen den Objekten. Berechnende Operationen sollten nur dann modelliert werden, wenn die Spezifikation eine Implementation der Berechnung vorgeben soll. Im Regelfall können berechnende Operationen durch simple Nachrichten ersetzt werden, die ledeglich vorgeben, dass ein gewisser Rechenschritt erfolgreich durchgeführt werden muss. Welche Informationen zwischen den Objekten ausgetauscht werden, können wir dem Agent Diagram entnehmen.

Nehmen wir zum Beispiel den Datenaustausch zwischen dem *SystemController* und dem *TemperatureSensor*. Der *SystemController* beobachtet das Attribut *Reading* des *TemperatureSensors*. Ein typisches Muster um eine solche beobachtende Beziehung zwischen zwei Agenten abzubilden ist, eine *request()* Nachricht von dem beobachtenden Agenten zu senden und von dem beobachteten Agenten mit einer *respond()* Nachricht zu antworten. In unserem Fall fügen wir dementsprechend eine *requestReading()* Operation zu dem *TemperatureSensor* und eine *respondReading()* Operation zu dem *SystemController* hinzu. An dieser Stelle stellt sich die Frage, ob wir einen Rückgabewert für die *respondReading()* Operation hinzufügen sollen. Eine gute Faustregel ist es, erst einmal mit gut benannten Operationen anzufangen und Parameter erst dann hinzuzufügen, wenn bestimmte Sachverhalte in der Spezifikation nur mit konkreten Werten abgebildet werden können.

Das vollständige Klassenmodell für das konzeptuelle Objekt *Heater* könnte zum Beispiel so aussehen wie in Abbildung 4.2.

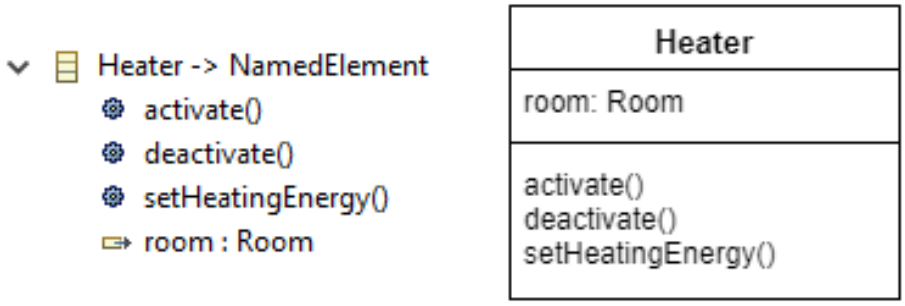

Abbildung 4.2: Klassenmodell - Die Klasse *Heater* mit drei Operationen und einer Referenz. EMF Ecore Darstellung Links und UML Darstellung rechts.

So verfahren wir nun für alle konzeptuellen Objekte in dem Object Model, sodass wir ein vollständiges Klassenmodell für unser System erhalten.

Anzumerken ist, dass das Klassenmodell während des Schreibens der SML Spezifikation erfahrungsgemäß noch geändert werden muss, um bestimmte Szenarios vernünftig modellieren zu können.

Das finale Klassenmodell befindet sich in Anhang B.

4.3 Erstellen der Dynamischen Instanz des System Containers

Nachdem wir das Klassenmodell erstellt haben, benötigen wir eine Dynamische Instanz des System Containers, welche als Objektmodell für unser Projekt dienen wird. Die Dynamische Instanz des System Controllers sollte dabei alle Systemobjekte beinhalten, die in der SML Spezifikation benötigt werden.

In unserem Fall sollte die Dynamische Instanz des System Controllers ein vollständiges *Apartment*, einen *SystemController*, die *HeatingFormula*, einen *Timer* und einen *User* beinhalten. Bei dem *Apartment* ist darauf zu achten, dass wir dem *LivingSpace* mindestens zwei Räume zuweisen, damit wir die Systeminteraktion mit mehreren Räumen im Wohnraum testen können. Alle Räume benötigen außerdem einen *TemperatureSensor* und einen *Heater*. Um später bei der Simulation einen einfacheren Überblick über die unterschiedlichen Objekte zu haben, vergeben wir außerdem eindeutige Namen für jedes Objekt in unserer Dynamischen Instanz.

Die vollständige Dynamische Instanz unseres *HomeautomationSystems* wird in Abbildung 4.3 gezeigt.

Sollten Änderungen an dem Klassenmodell vorgenommen werden, muss eventuell auch die Dynamische Instanz des System Containers angepasst werden, denn nur Objekte, die in der Dynamischen Instanz instanziert werden, können in der SML Spezifikation genutzt werden.

Die finale Version der Dynamischen Instanz des *HomeautomationSystems* befindet sich in Anhang B.

4.4 SML Grundlagen

Die SML Spezifikation ist der wichtigste Teil einer szenariobasierten Spezifikation in ScenarioTools. SML ist eine domänenspezifische Sprache, die auf Basis der Life Sequence Charts entworfen wurde. SML bietet die Möglichkeit, Interaktionen zwischen den Systemobjekten nachrichtenbasiert zu beschreiben. Systemobjekte werden in SML in kontrollierbare Objekte (Software) und unkontrollierbare Objekte (Environment) unterschieden [8].

Eine Nachricht an ein anderes Objekt wird in SML ein **Event** genannt. Das Schicken einer Nachricht an ein Objekt wird als Auslösen eines Events bezeichnet. Das nächste erwartete Event in einem Szenario wird aktiviertes Event genannt. Events können durch die Schlüsselwörter *commited*, *urgent*

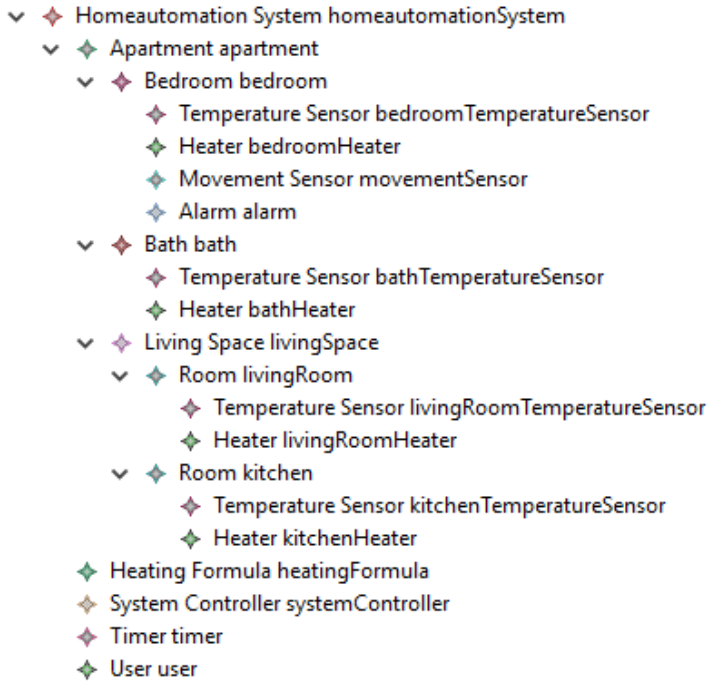

Abbildung 4.3: Die Dynamische Instanz des *HomeautomationSystems*

und *eventually* mit **Liveness Conditions** versehen werden. Jedes Event, das mit einer Liveness Condition versehen wurde, darf nicht auf ewig aktiviert bleiben. Wie schnell ein Event ausgelöst werden muss, hängt von dem benutzten Schlüsselwort ab. Events, die als *committed* oder *urgent* markiert wurden, müssen möglichst zeitnah ausgelöst werden, wobei nur andere *committed* und *urgent* Events vorher ausgelöst werden dürfen. Events, die als *committed* markiert wurden, haben Vorrang for *urgent* Events. Events, die als *eventually* markiert wurden, können zu einem beliebigen Zeitpunkt in der Zukunft ausgelöst werden. Zusätzlich zu den Liveness Conditions gibt es noch das Schlüsselwort *strict*, welches ausdrückt, dass das markierte Event in genau der Reihenfolge ausgelöst werden muss, wie es in dem Szenario auftaucht. Wird ein als *strict* markiertes Event zu einem falschen Zeitpunkt ausgelöst, bricht das Szenario mit einer **Safety Violation** ab. Ist ein Event nicht als *strict* markiert, wird das Szenario zwar auch abgebrochen, jedoch nicht verletzt [8].

Um ein Systemobjekt in einer SML Spezifikation zu nutzen, muss dem entsprechenden Objekt in der Spezifikation eine Rolle zugewiesen werden. Dieses Konzept funktioniert ähnlich den Lebenslinien in einem UML Sequenzdiagramm [8]. **Statische Rollen** beschreiben genau ein einzelnes Objekt in dem System. Ein einzelnes Objekt kann dabei unterschiedliche Rollen gleichzeitig einnehmen. Die Zuweisung von statischen Rollen geschieht in

der Runconfiguration. **Dynamische Rollen** hingegen werden in einem Szenario bei dessen Aktivierung von der ScenarioTools Runtime dynamisch gebunden. Ein Szenario kann dabei mehrfach gleichzeitig aktiv sein, aber nur, wenn sich die an die Rollen gebundenen Objekte unterscheiden. In unserem Fallbeispiel wären *Room*, *TemperatureSensor* und *Heater* zum Beispiel Kandidaten für eine dynamische Rolle, da das Heizen eines Raumes an keinen bestimmten Raum gebunden ist, sondern für alle Räume gleich funktioniert. **Dynamische Multi Rollen** sind eine bestimmte Art der dynamischen Rolle, die für Collections in dem Objektmodell genutzt werden sollten. Dynamische Multi Rollen werden, wie normale dynamische Rollen, zur Laufzeit von der ScenarioTools Runtime gebunden und erzeugen eine aktive Szenario Instanz pro Eintrag in der zugrunde liegenden Collection.

Systemobjekte, die als *controllable* markiert wurden, können keine spontanen Nachrichten versenden. Das bedeutet, dass die Software nur dann Events auslösen kann, wenn das Event in einem Szenario aktiviert wurde, welches selbst durch ein Environment Event aktiviert wurde. Alle Objekte, die nicht als *controllable* markiert wurden, können Events spontan auslösen. Sollte das für bestimmte Events nicht erwünscht sein, können die entsprechenden Events als *non-spontaneous* markiert werden. *Non-spontaneous* Events können nur ausgelöst werden, wenn sie in einem *Assumption-Szenario* aktiviert sind [8].

Szenarios werden unterteilt in **Guarantee-Szenarios** und **Assumption-Szenarios**. *Guarantee-Szenarios* beschreiben ein gewünschtes Systemverhalten. Das erste Event in einem Szenario aktiviert das Szenario, sobald es ausgelöst wird. *Assumption-Szenarios* beschreiben die Annahme, wie sich die Umwelt nach einem bestimmten Event verhalten wird. Ein wiederkehrendes Muster ist es, *guarantee scenarios* durch ein Environment Event zu aktivieren und das gewünschte Systemverhalten inklusive notwendiger weiterer Environment Events zu beschreiben. Es ist jedoch auch möglich, ein *guarantee scenario* durch ein Software Event zu aktivieren, um so große Szenarios aufzusplitten oder Szenarios mit dynamischen Rollen zu isolieren.

Neben den Events gibt es in SML noch **Kontrollfluss-Schlüsselwörter**, wie zum Beispiel *wait* zum Warten auf eine Bedingung, *while* zum Wiederholen von Event-Sequenzen, *alternative* für Alternativen in einem Szenario und *parallel* für parallel ablaufende Event-Sequenzen [8]. Zusätzlich gibt es noch **Constraints**, die Einschränkungen für ein ganzes Szenario beschreiben. Dazu gehören unter anderem *forbidden* für verbotene Events, *interrupt* für Events, die das Szenario (ohne Violation) unterbrechen sollen sowie *ignore* für Events, die das Szenario bei falscher Sendereihenfolge nicht abbrechen sollen. Sollte eine Bedingung an einer bestimmten Stelle eines Szenarios einen Abbruch oder eine Violation bewirken, können die Inline-Schlüsselwörter *interrupt* und *violation* genutzt werden.

4.5 Erstellen der SML Spezifikation

Beim Erstellen der SML Spezifikation für ein System, welches mit KAOS modelliert wurde, ist das Wichtigeste, dass jedes funktionale Ziel aus dem Goal Diagram durch ein Szenario oder eine Kombination von Szenarios abgedeckt wird. Das kann erreicht werden, indem die beschriebenen Szenarios eine Menge von verfeinerten Zielen beschreiben, die ebenfalls die Erfüllung aller überliegenden Ziele garantieren. Soft Goals und nicht-funktionale Ziele, wie Anforderungen an das Budget oder die Performance, können nicht modelliert werden.

Nehmen wir zum Beispiel das Top-Level Ziel *Maintain*[HealthyClimateInLivingSpaceDuringUse]. Wir suchen nun nach einer Menge von verfeinerten Zielen, die sich gut als Szenario beschreiben lassen und gleichzeitig das betrachtete Top-Level Ziel sowie alle dazwischenliegenden Ziele erfüllen. Die Ziele *Achieve*[StartHeating30MinutesBeforeWakeupTime] und *Maintain*[HeatRoomsWithActiveHeaterTo22Degrees] sind gute Kandidaten für ein Szenario, da sie einen bestimmten Interaktionsfluss beschreiben, aber nicht zu umfangreich sind. Wir benötigen außerdem noch das Ziel *Achieve*[WakeupTimeConfiguredByUser], um eine vollständige Abdeckung aller überliegenden Ziele zu erreichen.

Um das Ziel *Achieve*[StartHeating30MinutesBeforeWakeupTime] als Szenario zu beschreiben, müssen wir uns überlegen, welches Event das Szenario aktivieren soll und welche Interaktionen auf das auslösende Event folgen müssen, um das beschriebene Ziel zu erfüllen. Die verfeinerten Ziele des Goals Diagrams sowie die Informationen aus dem das Agent Diagram und dem Operationalization Diagram können uns dabei helfen, die richtigen Systeminteraktionen zu identifizieren. Das verfeinerte Ziel *Achieve*[Schedule30MinutesBeforeWakeupTime] deutet darauf hin, dass das auslösende Event das Signal *ThirtyBeforeScheduledTime* des *Alarms* ist. Das Ziel *Achieve*[StartLivingSpaceHeaters] gibt uns Informationen über die Folge-Events.

Wir modellieren diese Informationen in einem *guarantee scenario* namens *StartHeating30MinutesBeforeWakeupTime*, welches als eingehendes Event das Signal 30 Minuten vor der Weckzeit erhält. Danach muss der *SystemController* alle Heizungen des Wohnraumes anschalten. Wir erreichen dies, indem wir für das Szenario eine Multi Rolle *multiRoom* binden, die auf alle Räume des Wohnraumes verweist. Dadurch wird ein Szenario pro Raum in Wohnraum aktiviert, sobald das auslösende Event empfangen wurde. Das Anschalten der Heizung erfolgt durch eine Nachricht and die dynamische Rolle *Heater*, die an das Heizelement der Multi Rolle gebunden wird. Um zu garantieren, dass der *SystemController* zeitnah antwortet, markieren wir das Anschalten der Heizung als *urgent*. Listing 1 zeigt das resultierende Szenario.

Das Ziel *Maintain*[HeatRoomsWithActiveHeaterTo22Degrees] ist ein wenig komplizierter zu modellieren. Es lässt sich schnell erkennen, dass das

```
1 guarantee scenario StartHeating30MinutesBeforeWakeupTime
2 bindings[
3     multiRoom = livingSpace.rooms
4     heater = multiRoom.heater
5 ]{
6     alarm -> controller.signalThirtyMinutesBeforeScheduledWakeupTime()
7     urgent controller -> heater.activate()
8 }
```

Listing 1: SML Spezifikation - StartHeating30MinutesBeforeWakeup

Aktivieren eines *Heaters* durch den *SystemController* als auslösendes Event dient. Um den Raum zu heizen, muss der *SystemController* die aktuelle Temperatur des Raumes, in dem der Heizkörper steht, von dem Temperatursensor erfahren, die Information an die *HeatingFormula* weiterreichen und die errechnete Heizleistung an den Heizkörper zurücksenden. Das ganze soll außerdem so lange wiederholt werden, bis der Heizkörper abgeschaltet wurde.

In SML beschreiben wie dieses Ziel in einem neuen *guarantee scenario* namens *HeatRoomsWithActiveHeaterTo22Degrees*. Da das Szenario für Heizkörper in beliebigen Räumen gilt, binden wir den Raum des Heaters und den zugehörigen Temperatursensor an die Rollen *room* und *temperatureSensor*. Da das Heizen in einer Endlosschleife laufen soll, die nur dann abgebrochen wird, wenn der *Heater* deaktiviert wird, fügen wir eine *while* Schleife ohne Bedingung hinzu und setzen das Deaktivieren des Heizelements als *interrupt* Bedingung in den Constraints des Szenarios. Als nächstes müssen wir den Austausch der Informationen zwischen *SystemController*, *TemperatureSensor* und *HeatingFormula* beschreiben. Für den Informationsaustausch zwischen dem Controller und dem Temperatursensor benutzen wir eine Kombination aus request und respond Events. Das request Event an den Temperatursensor markieren wir außerdem als *eventually*, um zu modellieren, dass der *SystemController* entscheidet, wann einer neuer Heizzyklus beginnt.

Der Informationsaustausch zwischen Controller und Heizformel ist ein wenig komplizierter. Sowohl der *SystemController* als auch die *HeatingFormula* sind statische Systemteilnehmer, so dass wir ein Identifikationsmerkmal benötigen, um die berechneten Heizwerte einem Raum zuzuordnen. Glücklicherweise besitzt jeder *Room* ein Attribut *roomId*, welches wir dafür nutzen können. Der Austausch erfolgt wieder über request und respond Events, die zusätzlich die *roomId* als Parameter übergeben. Zum Schluss muss noch die berechnete Heizleistung an den *Heater* zurückgesendet werden. Das request/respond Muster für den Temperatursensor und die Heizformel wird in zwei weiteren *assumption scenarios* implementiert. Listing 2 zeigt das resultierende Szenario.

```
1  guarantee scenario HeatRoomsWithActiveHeaterTo22Degrees
2  bindings[
3      room = heater.room
4      temperatureSensor = room.temperatureSensor
5  ]{
6      controller -> heater.activate()
7      var EInt roomId = room.roomId
8      while {
9          systemInit -> controller.newTick()
10         eventually controller -> temperatureSensor.requestReading()
11         committed temperatureSensor -> controller.respondReading()
12         committed controller -> heatingFormula.requestNeededEnergyForTemperature(
               roomId)
13         committed heatingFormula -> controller.respondNeededEnergyForTemperature(
               roomId)
14         committed controller -> heater.setHeatingEnergy()
15     }
16 } constraints [
17     interrupt controller -> heater.deactivate()
18     ignore controller -> heatingFormula.requestNeededEnergyForTemperature(*)
19     ignore heatingFormula -> controller.respondNeededEnergyForTemperature(*)
20     ignore systemInit -> controller.newTick()
21 ]
22
23 assumption scenario TemperatureSensorDeliversAccurateMeasurementsForRoom
24 bindings[
25     room = temperatureSensor.room
26 ]{
27     controller -> temperatureSensor.requestReading()
28     temperatureSensor -> controller.respondReading()
29 }
30 assumption scenario HeatingFormulaCalculatesHeatingEnergy {
31     var EInt roomId
32     controller -> heatingFormula.requestNeededEnergyForTemperature(bind roomId)
33     heatingFormula -> controller.respondNeededEnergyForTemperature(roomId)
34 }
```

Listing 2: SML Spezifikation - HeatRoomsWithActiveHeaterTo22Degrees

Anmerkung: In dem resultierenden Szenario wurden einige Tricks benutzt, um die Synthese zu beschleunigen und kleine Eigenheiten von SML zu umgehen. Zum einen wurde ein neues Environment Event an den Anfang der Schleife gefügt, um bei der Synthese einen Zustand zu erzeugen, in dem das System keine Liveness Constraints erfüllen muss. Dies ist eine Eigenheit des genutztes Synthese-Algorithmus und spielt für die Simulation keine Rolle. Zum anderen wurden alle Events nach dem ersten request Event als *committed* markiert, um die Anzahl der generierten Zustände während der Synthese drastisch zu verringern. Zuletzt wurde für einige Events ein *ignore* Constraint hinzugefügt, da in SML bei Events mit Parametern nicht zwischen tatsächlichen Parameterwerten unterschieden wird. Als Workaround bietet es sich an, ein weiteres Senden des selben Events nicht als Abbruchbedingung des Szenarios zu werten.

Das Ziel *Achieve*[WakeupTimeConfiguredByUser] ist sehr einfach in einem Szenario zu beschreiben, da lediglich eine Nachricht für das Einstellen der Weckzeit vom *User* zum *Alarm* gesendet werden muss. Zusätzlich

hängen an dem Event keine weiteren Interaktionen, da der *Alarm* dem *SystemController* Bescheid gibt, sobald der Zeitpunkt 30 Minuten vor der Weckzeit erreicht wurde. Tatsächlich ist ein so simples Szenario ein Indiz dafür, dass dieses Ziel eventuell besser als Assumption modelliert werden sollte. Wir verwerfen also das Szenario und vermerken die Erkenntnis für die nächste Iteration der KAOS-Modellüberarbeitung.

So verfahren wir nun für die restlichen Ziele des Goal Models, bis wir eine vollständige Abdeckung durch die Szenarios in der SML Spezifikation erreichen. Abschließend muss noch die Runconfiguration erstellt werden, die die statischen Rollen der Spezifikation den Objekten in der Dynamischen Instanz des System Containers zuordnet.

Die finale SML Spezifikation sowie die zugehörige Runconfiguration können in Anhang B gefunden werden.

4.6 Simulation und Synthese der Spezifikation

Nachdem wir alle Teile der szenariobasierte Spezifikation erstellt haben, können wir das System in ScenarioTools simulieren und eine Controller Synthese durchführen.

ScenarioTools nutzt für die Simulation und die Controller Synthese den Play-Out-Algorithmus von Harel et al. [11, 12], welcher definiert, welche Event-Sequenzen in einem System zulässig sind, ohne dass die szenariobasierte Spezifikation verletzt wird. Der Algorithmus wartet darauf, dass die Umwelt ein Event sendet, aktiviert alle entsprechenden Szenarios, setzt bereits aktive Szenarios weiter und wählt eine Menge von Reaktionen (in Form von Events), die für alle aktiven Szenarios gültig sind. Sollte keine Reaktion gültig sein, jedoch Liveness Constraints bestehen, gilt die Spezifikation als nicht erfüllt [8].

Bei der Simulation spielt der Benutzer die Umwelt und das System, indem er die Events auslöst, die der Play-Out-Algorithmus als zulässig einstuft. So können während der Simulation manuell die Hauptinteraktionen des Systems durchgespielt werden.

Bei der Controller Synthese erforscht ScenarioTools automatisch alle Zustände und Transitionen des Systems und präsentiert dem Benutzer eine Strategie für das Erfüllen der Spezifikation (falls möglich) sowie einen vollständigen Zustandsgraphen. Für das Erforschen aller Zustände und Transitionen wurde in dieser Arbeit der GR(1) Synthese-Algorithmus von Chatterjee at al. [4] benutzt. ScenarioTools bietet jedoch noch weitere Synthesealgorithmen an.

Es ist ebenfalls möglich, eine SML Spezifikation zur Laufzeit zu interpretieren und so ein System zu steuern [7], oder ausführbaren Code aus einer Spezifikation zu generieren [8]. Je nach Größe des spezifizierten Systems und Detailgrad der Spezifikation kann dies jedoch sehr aufwendig sein.

Für das Erstellen der finalen szenariobasierten Spezifikation in Anhang B haben sich sowohl Simulation als auch Controller Synthese als wertvolle Werkzeuge erwiesen, um Fehler und Unstimmigkeiten in der Spezifikation zu finden. Zusätzlich fördern beide Werkzeuge einen iterativen Entwicklungszyklus beim Schreiben der Spezifikation, was zu einem gut durchdachten Endprodukt führt. Aus Platzgründen wird in dieser Arbeit jedoch nicht weiter auf die Simulation und Controller Synthese eingegangen.

4.7 Zusammenfassung

In diesem Kapitel wurde vorgestellt, wie eine szenariobasierte Spezifikation systematisch aus einem KAOS System Model abgeleitet werden kann. Die präsentierten Techniken wurden erfolgreich anhand des Fallbeispiels aus Kapitel 3 umgesetzt. Die resultierende szenariobasierte Spezifikation kann in Anhang B gefunden werden.

Das folgende Kapitel beschäftigt sich mit weiterführenden Techniken, die dabei helfen, von einer szenariobasierten Spezifikation zu einer Implementation des System-to-be zu gelangen.

Kapitel 6 diskutiert den in Kapitel 3 und 4 präsentierten Ansatz für den systematischen Entwurf einer szenariobasierten Spezifikation.

Kapitel 5

Weiterführende Techniken

In diesem Kapitel wird gezeigt, wie das Benutzen einer *Ubiquitous Language* bei der Überwindung von Sprachbarrieren im Projekt helfen kann und warum sich das *Actor Model* für die Programmierung eines Systems anbietet, für das eine szenariobasierte Spezifikation erstellt wurde.

5.1 Ubiquitous Language

Der Begriff Ubiquitous Language stammt aus dem Buch *Domain Driven Design* von Eric Evans [5] und bezeichnet ein domänennahes Vokabular, das von jeder Partei, die an dem Projekt beteiligt ist, genutzt wird.

Das domänennahe Vokabular ergibt sich aus den Gesprächen mit dem Kunden, den Domänenexperten und den Benutzern des Systems. Ziel ist es, dass alle Stakeholder des Produktes die selbe Sprache sprechen und sprachliche Barrieren zwischen Domänenexperten und Entwicklern durchbrochen werden.

Nehmen wir zum Beispiel das Home Automation Fallbeispiel aus Kapitel 3. Die Beschreibung des Systems aus der Elicitation Phase bezeichnet die Zielgruppe des Systems als Benutzer. Der Begriff "User" ist ein gängiges Konzept in der IT-Branche, weshalb es allen an dem Projekt Beteiligten leicht fallen sollte, das Konzept eines Benutzers zu verstehen. Hätte sich die fiktive Firma SmartHome jedoch dazu entschieden, anstelle von einem Benutzer von einem "Bewohner" zu sprechen, könnten sich die Entwickler des Systems dazu entscheiden, das selbe Konzept in dem Programm trotzdem "User" zu nennen, anstelle von einer passenderen Übersetzung, wie "Resident", oder "Occupant". Treffen sich nun der Kunde und das Entwicklerteam, stehen beide Seiten vor einer sprachlichen Barriere, da der Kunde von einem Bewohner und das Entwicklerteam von einem Benutzer sprechen.

Mag eine Sprachbarriere bei einem einzelnen Konzept noch leicht zu überwinden sein, wird das Problem exponentiell schlimmer, um so

gespaltener das Vokabular der unterschiedlichen beteiligten Gruppen in dem Projekt ist. Bei einer Besprechung zweier Gruppen muss jede Gruppe von dem eigenen Vokabular in ein Vokabular übersetzen, welches sie nie genutzt hat. Dieser Übersetzungsprozess ist zeitaufwendig und provoziert Missverständisse und Fehler.

Ein gemeinsamer Wortschatz beugt solchen Problemen vor und sorgt dafür, dass sich die Entwickler tiefer mit der Domäne befassen. Während der Erklärung des KAOS Object Model in Kapitel 3.4 wurde bereits erwähnt, dass es sich lohnt, ein Definitions-Glossar während der Anforderungsanalyse anzufertigen, das gängige Domänenkonzepte eindeutig beschreibt. So ist es schon während der Anforderungsphase möglich, ein gemeinsames, wohldefiniertes Vokabular zu erarbeiten.

In dieser Arbeit wurde ebenfalls eine Ubiquitous Language genutzt. Die Formulierungen aus der Beschreibung des Systems in der Elicitation Phase ziehen sich durch die KAOS Modelle sowie die szenariobasierte Spezifikation. Das erleichtert es, die Zusammenhänge der Modelle und der Spezifikation zu verstehen und gibt zusätzlich ein domänennahes und allgemein genutztes Vokabular für alle nachfolgenden Aktivitäten, wie die Architektur, die Entwicklung, das Testen und die Dokumentation.

Zusammenfassend beschreibt das Konzept der Ubiquitous Language die Nutzung eines gemeinsamen, domänennahen Vokabulars von allen beteiligten Parteien eines Projektes. Das erleichtert die Kommunikation zwischen den unterschiedlichen Teilbereichen und ermöglicht es den Domänenexperten, einen Überblick über das Produkt zu haben, ohne mit einem fremden oder technischen Jargon konfrontiert zu werden.

5.2 Actor Model

Das Actor Model beschreibt ein mathematisches Modell für parallele Programmierung, bei dem sogenannte Actors als Hauptwerkzeug für die Verarbeitung von Daten, das Verwalten von Zustand und den Austausch von Informationen genutzt werden.

Ein Actor ist ähnlich einem Objekt in der objektorientierten Programmierung. Jeder Actor besitzt einen Zustand und kann Berechnungen durchführen. Im Gegensatz zu der objektorientierten Programmierung arbeiten alle Actors eines Systems jedoch parallel und tauschen Informationen über Nachrichten aus und nicht über Methodenaufrufe. Der Zustand jedes Actors ist dabei nicht öffentlich einsehbar, sondern muss über den Austausch von Nachrichten propagiert werden. Ein Programm, welches auf Basis des Actor Models entworfen wurde, kann deshalb sehr einfach auf mehrere Rechenknoten verteilt werden [2].

Es gibt mehrere erfolgreiche Implementationen des Actor Models. Die

bekannteste findet sich in *Erlang* [1], einer von Ericsson entwickelten Programmiersprache, die sich besonders für Soft-Realtime Anwendungen mit hoher Anzahl von parallelen Berechnungen eignet. Erlang wurde seit dem Release in 1986 weltweit für die Programmierung von Software für Telefonswitches bei Ericsson eingesetzt. Eine Implementation für die Java Virtual Machine bietet *Akka* [2], ein Toolkit für die Programmierung mit Actors, das als Teil der Programmiersprache *Scala* im Jahr 2006 veröffentlicht wurde.

Da eine szenariobasierte Spezifikation sich primär auf die Interaktion zwischen den einzelnen Systemobjekten konzentriert, bietet sie eine perfekte Grundlage für die Entwicklung eines verteilten Systems mit Hilfe des Actor Models. Jedes Objekt in der Spezifikation wird als ein Actor implementiert, der genau wie in der Spezifikation auch, nachrichtenbasiert Informationen mit den anderen Actors des System austauscht. Neben dem nachrichtenbasierten Austausch von Daten zeigt eine szenariobasierte Spezifikation außerdem auf, wie bestimmte Systemkomponenten zusammenhängen, welche Interaktionen verboten sind und in welchen Systemzuständen das System bis zum nächsten Environment Event warten kann. Das Actor Model eignet sich ebenfalls gut für die Implementation von reaktiven Systemen, da häufig mit mehreren autonomen Komponenten innerhalb des Systems parallel kommuniziert werden muss.

Das alles macht eine szenariobasierte Spezifikation zu einem sehr guten Spezifikationswerkzeug für Systeme, welche auf Basis des Actor Models implementiert werden können.

[1] www.erlang.org
[2] www.akka.io

Kapitel 6

Diskussion

In diesem Kapitel wird der in Kapitel 3 und 4 vorgestellte Ansatz zum systmatischen Entwurf einer szenariobasierten Spezifikation anhand eines KAOS System Models diskutiert. Zusätzlich werden verwandte Arbeiten vorgestellt und ein abschließendes Fazit gezogen.

6.1 Bewertung des Ansatzes

Wie wir bereits anhand des Home Automation Fallbeispiel gesehen haben, ist das systematische Erstellen einer szenariobasierten Spezifikation auf Basis eines KAOS System Models durchaus anwendbar. Obwohl kein vollständiges KAOS System Model erstellt wurde, ist das Ergebnis zufriedenstellend und die präsentierten Schritte ergeben eine aufeinander aufbauende Strategie, um von einer Systembeschreibung aus der Elicitation Phase zu einer vollständigen, simulier- und synthetisierbaren Spezifikation zu gelangen.

Jedoch verlief der Modellierungsprozess während des Schreibens der Arbeit nicht so linear, wie in der Arbeit präsentiert. Vielmehr zog sich der Prozess über mehrere Iterationen hinweg. So haben sich während dem Erstellen des Responsibility Models und des Operation Models mehrere Änderungen an dem Goal und Object Model ergeben, da Unstimmigkeiten in der Verantwortlichkeitsverteilung und dem Informationsfluss gefunden wurden. Änderungen an dem Goal Model haben andersherum größere Überarbeitungen der anderen Modelle erfordert.

Ebenso wurden viele Fehler an den KAOS Modellen gefunden, während die SML Spezifikation geschrieben wurde. Das Schreiben von SML und das Simulieren der Spezifikation offenbarte sich als ein stark logischer Denkprozess, bei dem bisher noch nicht berücksichtigte Randfälle entdeckt wurden, die rückwirkend in den KAOS Modellen mit einbezogen werden mussten. Der iterative Modellierungs- und Entwicklungsprozess führte dabei zu wesentlich robusteren und sogar einfacheren Modellen.

Obwohl das Endprodukt durch die vielen Iterationen stark verbessert

wurde, war das ständige Überarbeiten der Modelle und der Spezifikation sehr zeitaufwendig. Schon kleine Änderungen an dem Modell konnte mehrere Stunden Arbeit in Anspruch nehmen, da mehrere Diagramme überarbeitet werden mussten. Die Konsistenz der Modelle, das heißt die Korrektheit der geteilten Informationen, war ebenfalls ein Punkt, an dem selbst bei einem so kleinen Projekt, wie dem Fallbeispiel, Fehler gemacht wurden. Ein weniger formaler RE-Prozess wäre hier produktiver gewesen.

Ein generelles Problem des KAOS Frameworks ist außerdem die Größe der Modelle. Das Fallbeispiel für diese Arbeit wurde mehrfach vereinfacht, bis es im Rahmen der Arbeit aktzeptabel war. Trotzdem sind die resultierenden Diagramme in Anhang A so groß, dass sie nur auf mehreren Seiten abgebildet werden können. Selbst an einem großen Computerbildschirm war es schwierig, die Übersicht über das gesamte System Model zu behalten.

Ein Punkt, der in dieser Arbeit nur angeschnitten wurde, ist außerdem das Modellieren von nicht-funktionalen Anforderungen, wie Anforderungen an die Projektlaufzeit und das Budget sowie Performanceanforderungen und Sicherheitsanforderungen. Diese können zwar in KAOS modelliert werden, fühlen sich in den Modellen jedoch fehl am Platz an, da die Grenze zwischen funktionalen und nicht-funktionalen Zielen in den Modellen aufgrund der bereits erwähnten Größe der Modelle schnell verschwimmt. Es ist außerdem nicht möglich, solche Anforderungen in einer szenariobasierten Spezifikation festzuhalten. An dieser Stelle würde es sich anbieten, nicht-funktionale Anforderungen nicht in KAOS und einer szenariobasierten Spezifikation zu modellieren, sondern diese Anforderungen in einem separaten, natürlichsprachlichen Dokument zu notieren.

Das Ableiten der szenariobasierten Spezifikation aus den KAOS Modellen verlief reibungsfrei, was jedoch größtenteils dadurch kam, dass die Spezifikation direkt im Anschluss an die Modellierung mit KAOS entworfen wurde. Eine zeitliche Verzögerung zwischen den beiden Aktivitäten, oder ein Wechsel von verantwortlichen Personen, würde das Ableiten der Spezifikation erschweren. Zusätzlich gab es einige Eigenheiten von SML, die durch kleine Workarounds behoben werden mussten. Zum Beispiel gibt es in SML keine Ein-Mal-Nachrichten für die Initialisierung des Systems.

Abschließend sollte noch erwähnt werden, dass der in dieser Arbeit präsentierte Ansatz nicht besonders einsteigerfreundlich ist. Sowohl ScenarioTools als auch KAOS sind komplexe Werkzeuge, die eine Menge theoretisches Wissen und Praxiserfahrung benötigen, um ein gutes Endprodukt zu erhalten.

6.2 Verbesserungvorschläge

Ein Großteil der oben genannten Probleme hängen damit zusammen, dass das Entwerfen der Modelle sowie das Tracking zwischen den Modellen

komplett per Hand durchgeführt wurde. Eine Entwicklungsumgebung für KAOS könnte zum Beispiel eine Beschreibung der Modelle in einer domänenspezifischen Sprache in die entsprechenden Diagramme übersetzen und die Konsistenz zwischen den Modellen automatisch überprüfen.

Die Workarounds in SML hätten größtenteils vermieden werden können, wenn es neben den Tutorials auf der ScenarioTools Website eine ausführliche Dokumentation gegeben hätte.

6.3 Verwandte Arbeiten

6.3.1 Alternativen zu KAOS

In dieser Arbeit wurde für den präsentierten Ansatz KAOS genutzt. Es gibt jedoch noch weitere Frameworks, die genutzt werden könnten, um Informationen für eine szenariobasierte Spezifikation Goal-orientiert zu erarbeiten.

Das **NFR Framework** konzentriert sich auf das Modellieren und Analysieren von nicht-funktionalen Anforderungen. Dazu werden in dem Framework Soft Goal in einem AND/OR Graphen (ähnlich wie dem KAOS Goal Model) verfeinert und Abhängigkeiten zwischen Soft Goals dargestellt [13, 15].

i* ist ein Agenten-orientiertes Modellierungsframework, welches neben dem Requirements Engineering zusätzlich auch für das Modellieren von Geschäftsprozessen in Unternehmen, Impact Analysis der Unternehmensorganisation und das Modellieren von Softwareprozessen genutzt werden kann. Dazu analysiert i* alle Aktoren in einem System sowie ihre Goals, Abilities, Beliefs und Commitments. Um seine Ziele zu erfüllen, benötigt ein Aktor die Hilfe von anderen Aktoren. Diese Abhängigkeiten werden in dem sogenannten *Strategic Dependency Model* festgehalten. Das *Strategic Rationale Model* hingegen ist dafür da, um die Gründe für Prozesse in dem System und dem Unternehmen zu modellieren [13, 24].

Auf Basis des i* Frameworks wurde **Tropos** entwickelt, eine Requirement-getriebene, Agenten-orientierte Entwicklungsmethode, welche den Nutzer von der frühen Requirements Phase bis hin zum architekturellen Design des Systems unterstützt. Tropos nutzt i*, um Requirements zu modellieren und Systemkonfigurationsentscheidungen zu begründen [13, 3].

Die Goal Based Requirements Analysis Method (**GBRAM**) legt ihren Fokus auf die frühzeitige Identifikation und Abstraktion von Goals. Goals werden dabei aus bestehenden Informationsquellen wie Diagrammen, Beschreibungen und Interviews identifiziert. In einem separaten Refinement Prozess werden die Ziele dann, ähnlich wie bei KAOS, ausgearbeitet [13, 1].

Neben den anderen Goal-oriented Requirements Engineering Ansätzen

gibt es außerdem noch ein Softwaretool namens *Objectiver* [1], welches einen auf KAOS basierenden Ansatz zum Modellieren von Requirements bietet [17]. Das Tool wurde in dieser Arbeit aufgrund des Umfangs und der Lizenzkosten nicht weiter betrachtet.

6.3.2 Alternativen zu ScenarioTools

Neben ScenarioTools gibt es noch weitere Werkzeuge mit denen es möglich ist, das Verhalten von Systemen formal zu beschreiben.

Der **Play-In/Play-Out** Ansatz von Harel et al. [12] basiert auf der Idee das Verhalten eines Systems direkt aus den Interaktionen mit der Oberfläche des Systems (oder einer abstrakten Repräsentation) aufzuzeichnen (Play-In) und anschließend wiederzugeben (Play-Out). Der Play-Out-Algorithmus ist dabei der selbe, der auch bei der Simulation und Synthese in ScenarioTools genutzt wird. Ein Tool welches dieses Verfahren unterstützt ist *PlayGo*. PlayGo zeichnet die Interaktionen mit dem System in einer formalen, erweiterten Variante der Life Sequence Charts auf [10].

Behavioral Programming bezeichnet einen Ansatz für die Software Entwicklung, bei dem das Verhalten eines Systems in unterschiedlichen Verhaltenssträngen beschrieben wird, die zur Laufzeit des Systems so ineinander verwebt werden, dass sich ein integriertes Systemverhalten ergibt. Jeder Verhaltensstrang beschreibt eine Sequenz von Events, welche das System unter bestimmten Bedingungen ausführen oder nicht ausführen soll. Es existieren Implementationen für die Programmiersprachen Java, Erlang, C, C++ und Javascript sowie das Tool *PlayGo* [23].

6.4 Fazit

Der in dieser Arbeit präsentierte Ansatz bietet ein wirkungsvolles Verfahren, um eine szenariobasierte Spezifikation mit Hilfe des KAOS Frameworks zu entwerfen. Während dem Modellieren mit KAOS erlangt der Requirements Engineer einen tiefen Einblick in die Domäne des zu spezifizierenden Systems, was zusätzlich bei dem Entwurf der Spezifikation hilft. Im Gegensatz zu weniger formalen Verfahren ist das Modellieren mit KAOS jedoch recht aufwendig und benötigt solides Grundwissen und Praxiserfahrung, um ein gutes Endprodukt zu erhalten.

[1] www.objectiver.com

Kapitel 7

Zusammenfassung und Ausblick

In diesem Kapitel wird der Inhalt der Arbeit zusammengefasst und ein kurzer Ausblick gegeben, in welchen Bereichen auf Basis dieser Arbeit weiter geforscht werden könnte.

7.1 Zusammenfassung

In dieser Arbeit wurde ein Ansatz für den systematischen Entwurf von szenariobasierten Spezifikationen präsentiert.

In dem Grundlagenteil wurden die unterschiedlichen Phasen des Requirements Engineering präsentiert, eine Einführung in das Goal-oriented Requirements Engineering gegeben und die szenariobasierten Spezifikationen betrachtet.

Der Kern der Arbeit beschäftigt sich mit dem Erstellen eines Systemmodells mit Hilfe des KAOS Frameworks, einem Goal orientierten, modellgetriebenen Ansatz für das Erheben von Anforderungen und dem Entwurf einer szenariobasierten Spezifikation auf Basis der gesammelten Modellinformationen.

Im Detail wurden vier KAOS Modelle behandelt. Das Goal Model präsentiert die Ziele des Systems und verfeinert diese so lange, bis sie einem einzelnen Agenten in dem System zugewiesen werden können. Das Object Model präsentiert alle konzeptuellen Objekte des Systems inklusive ihrer Attribute und Assoziationen. Das Responsibility Model präsentiert die Verantwortlichkeiten der Agenten des Systems sowie ihre beobachteten und kontrollierten Attribute. Das Operation Model präsentiert eine funktionale Sicht auf das System und legt fest, welche Operationen ein Agent für die Erfüllung seiner Verantwortlichkeiten durchführen können muss.

Im Anschluss wurden die Informationen der KAOS Modelle in eine szenariobasierte Spezifikation überführt. Dazu wurde ScenarioTools, eine

Sammlungs von Eclipse Plugins genutzt, die es dem Benutzer ermöglichen, eine szenariobasierte Spezifikation mit formalen Mitteln direkt in Eclipse zu erstellen und überprüfen. Das Datenmodell der Spezifikation wurde direkt aus dem Goal Model abgeleitet und um Operationen ergänzt, die nachrichtenbasiert den Informationsaustausch innerhalb des Systems modellieren. Die SML Spezifikation wurde mit heuristischen Mitteln aus dem Goal Model, dem Responsibility Model und dem Operationalization Model abgeleitet.

Zum Schluss wurden zwei weiterführende Techniken präsentiert. Die Ubiquitous Language, eine Konzept aus dem Domain Driven Design, welches die Nutzung ein gemeinsamen, domänennahes Vokabulars fordert, und das Actor Model, ein mathematisches Modell für die Entwicklung von Systemen auf der Basis von autonomen Actors.

In der Diskussion wurden die Stärken und Schwächen des präsentierten Ansatzes herausgearbeitet und verwandte Arbeiten betrachtet.

7.2 Ausblick

Der in dieser Arbeit präsentierte Ansatz beschäftigt sich hauptsächlich mit dem Modellieren eines Systems mit Hilfe von KAOS. Interessant wäre es, andere Ansätze, wie zum Beispiel i* an Stelle von KAOS zu nutzen und die Verfahren zu vergleichen.

Ebenfalls wurde kein großer Fokus auf die Rolle des Ansatzes innerhalb des Requirements Lifecycle gelegt. Eine genaue Analyse, welche Phasen bearbeitet wurden und welche RE-Tätigkeiten vernachlässigt wurden, könnte Anreize geben, den präsentierten Ansatz zu verbessern.

Anhang A

KAOS Modelle

Anmerkung: Große Modelle wurden in mehrere Teilmodelle unterteilt. Drei Punkte am Ende einer Kante verweisen auf ein anderes Teilmodell. Breite Modelle werden aus Platzgründen quer abgebildet.

A.1 Goal Model

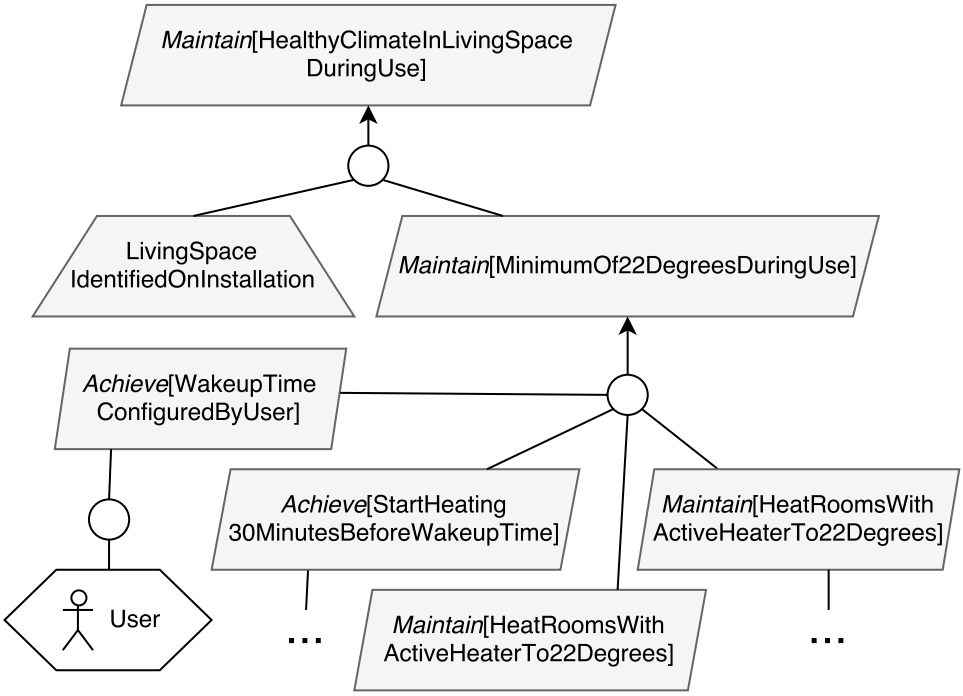

Abbildung A.1: Goal Diagram - Heizen des Wohnraums - Teil 1

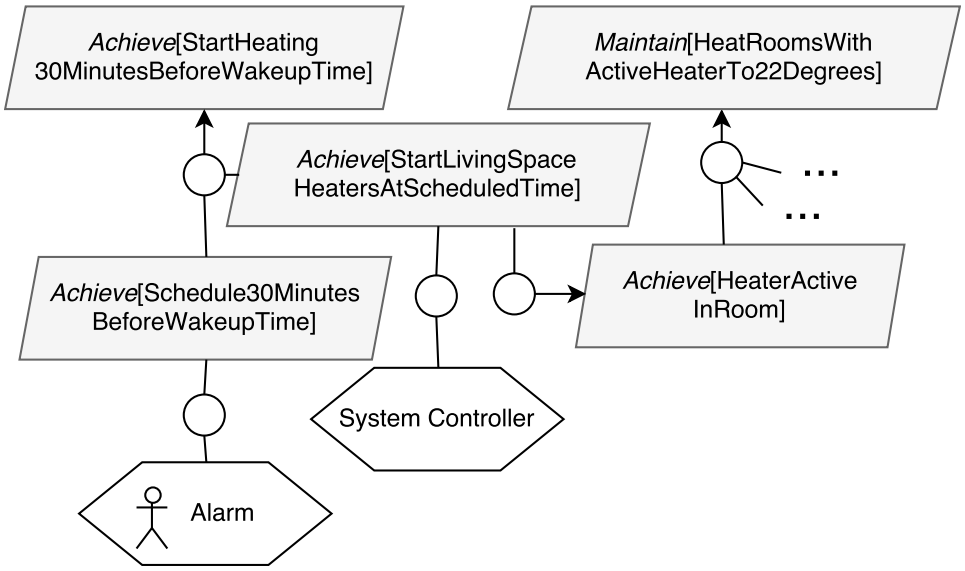

Abbildung A.2: Goal Diagram - Heizen des Wohnraums - Teil 2

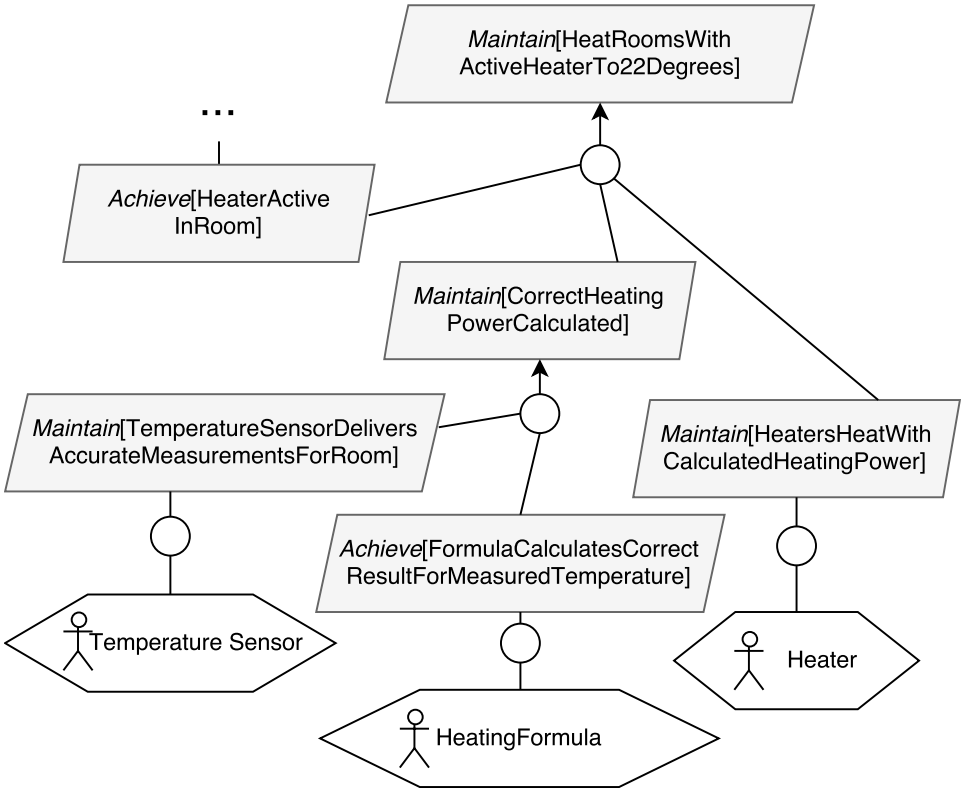

Abbildung A.3: Goal Diagram - Heizen des Wohnraums - Teil 3

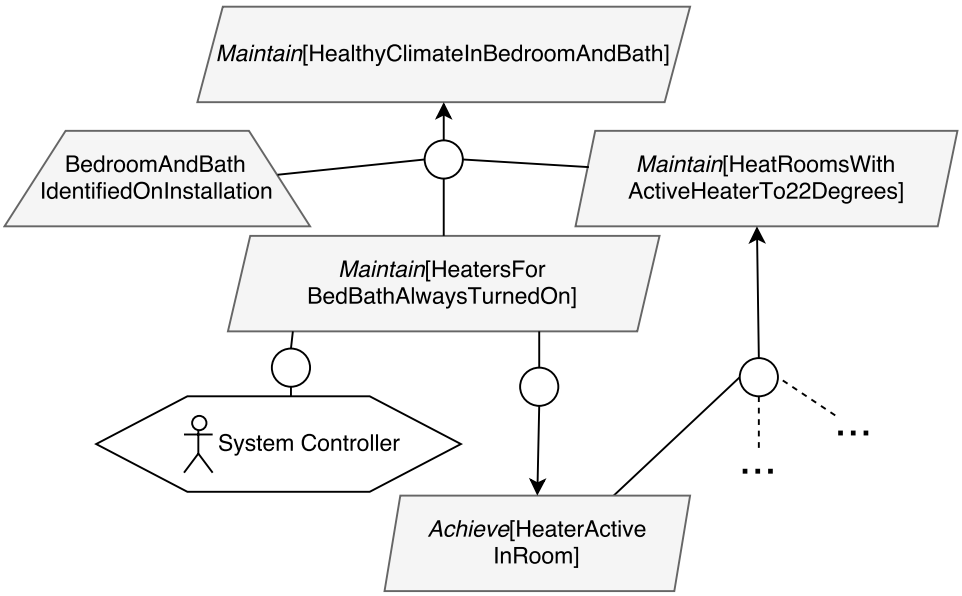

Abbildung A.4: Goal Diagram - Heizen von Schlafzimmer und Bad

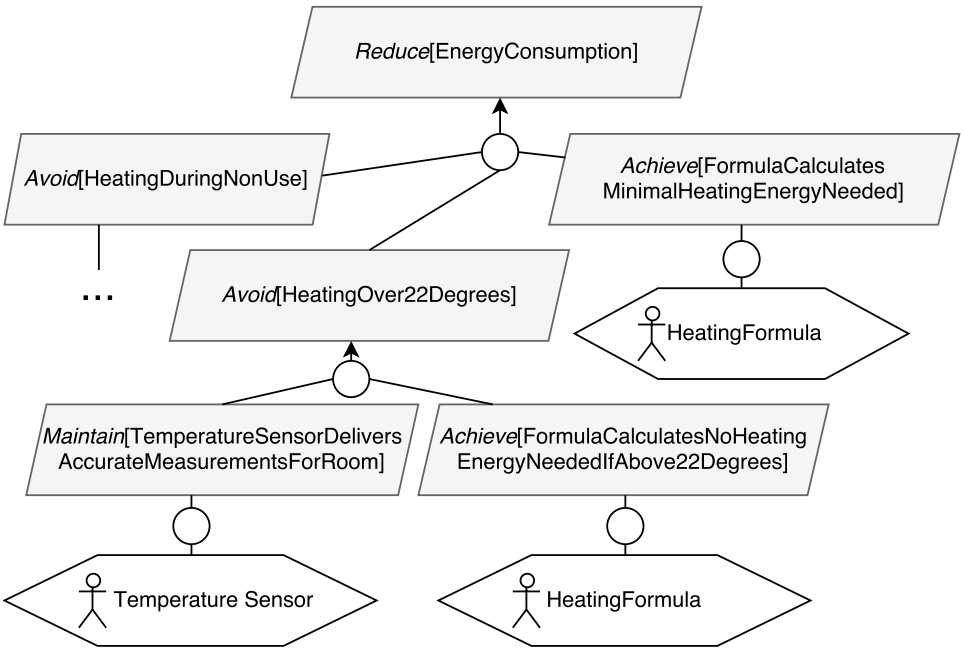

Abbildung A.5: Goal Diagram - Reduzierung des Energiebedarfs - Teil 1

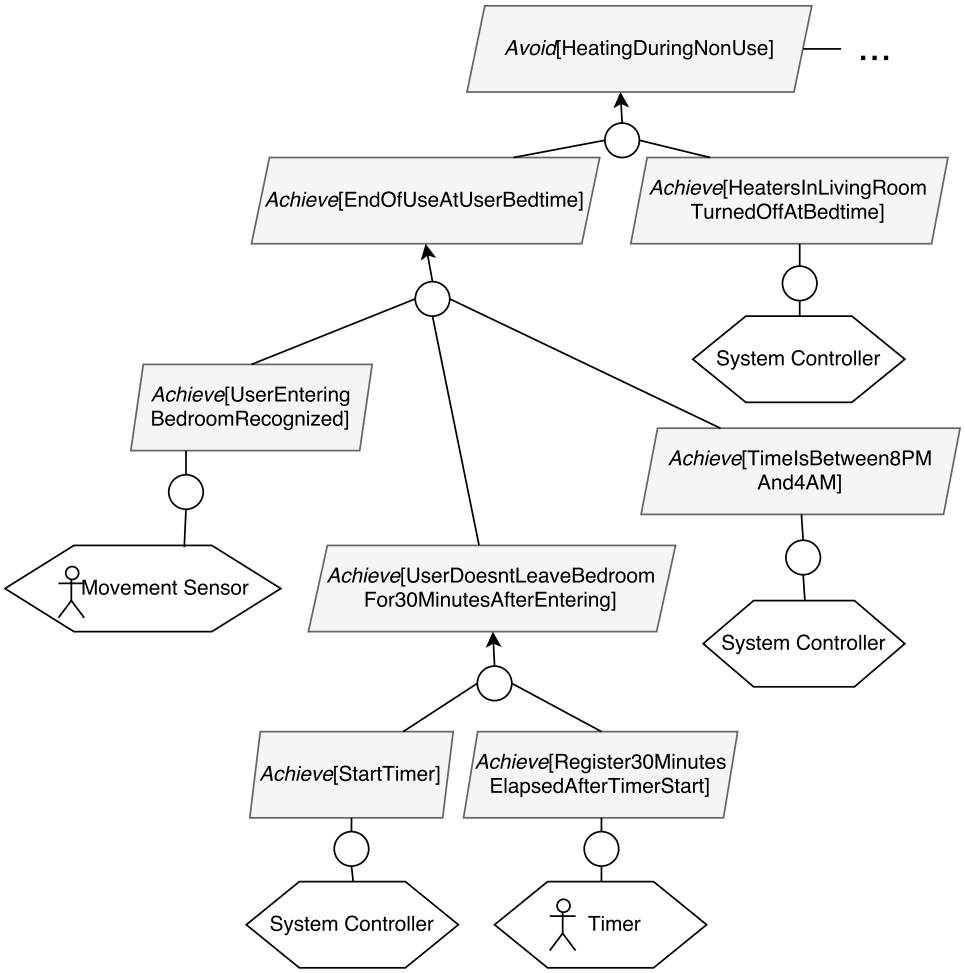

Abbildung A.6: Goal Diagram - Reduzierung des Energiebedarfs - Teil 2

A.2 Object Model

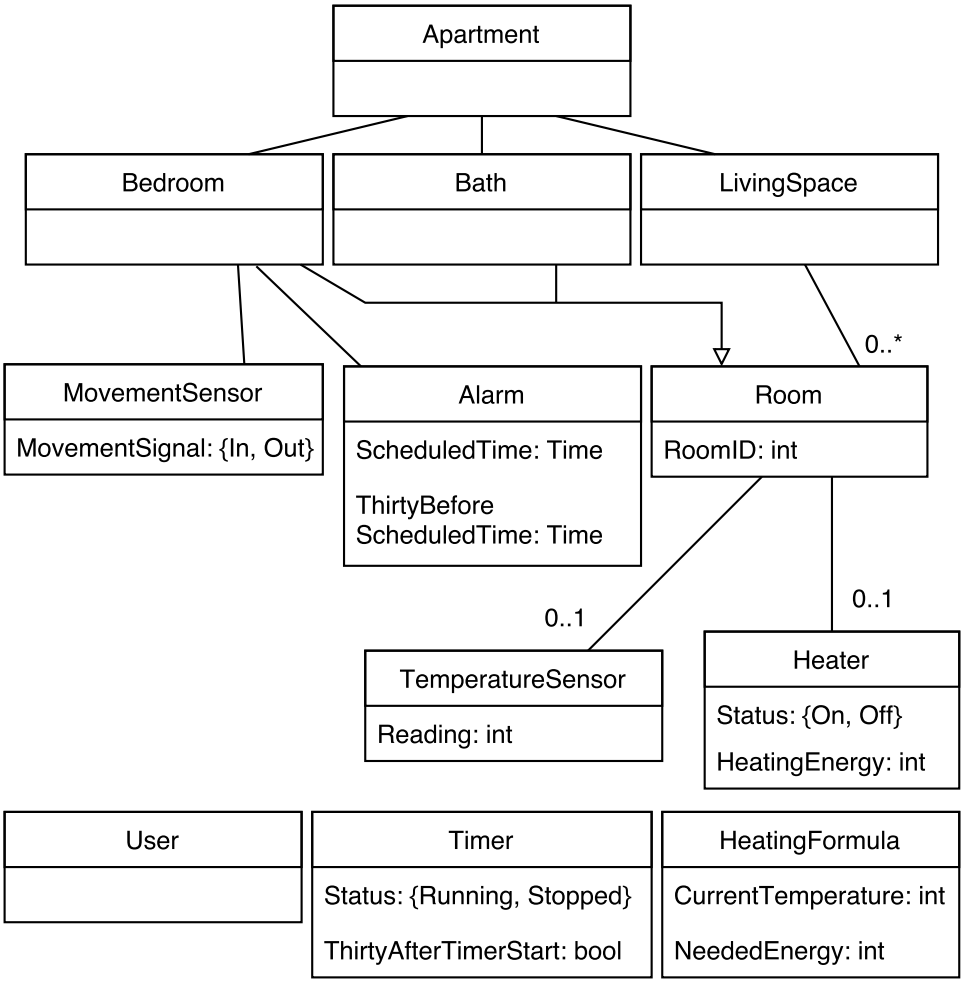

Abbildung A.7: Object Model

A.3 Responsibility Model

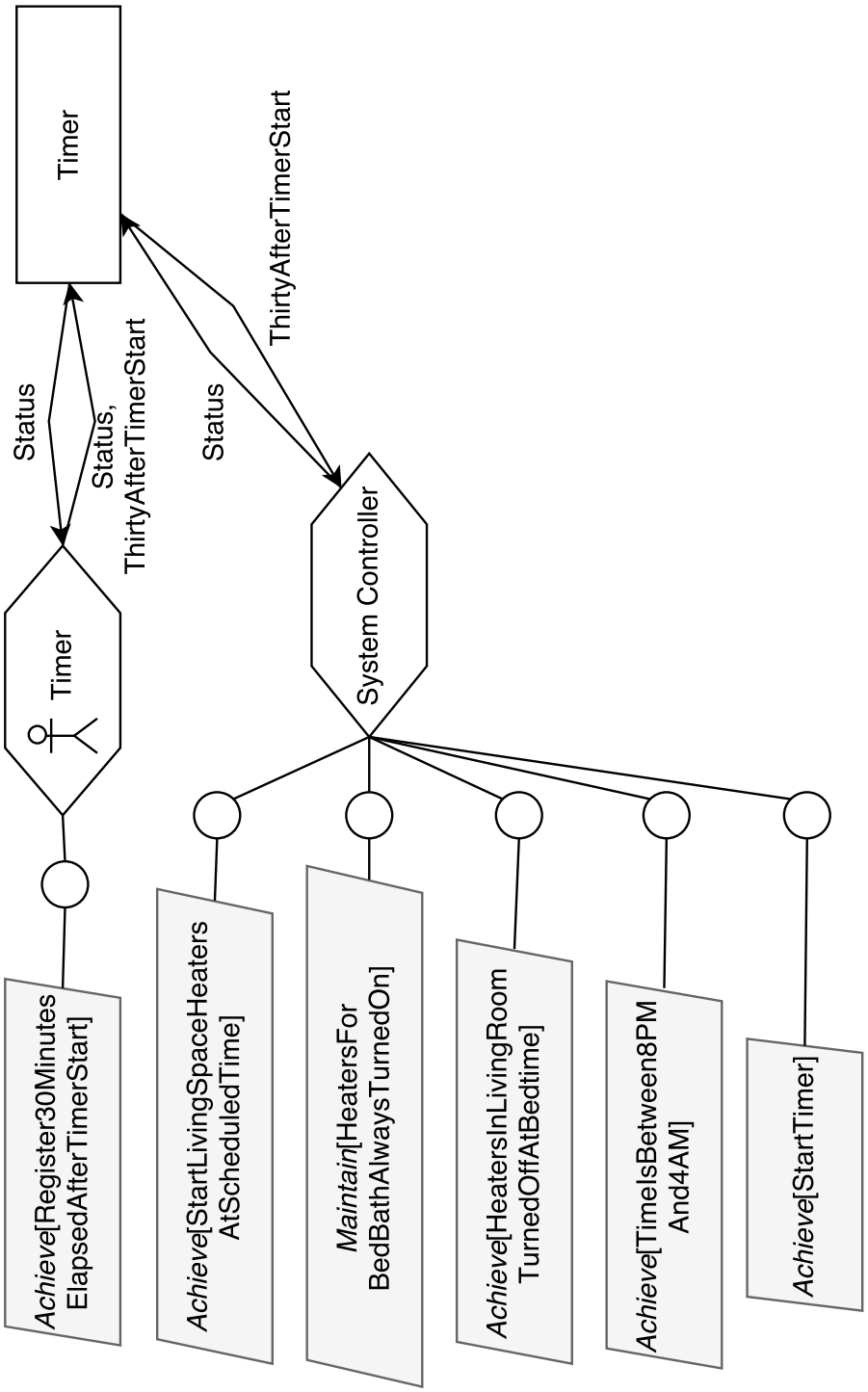

Abbildung A.8: Agent Diagram - Teil 1

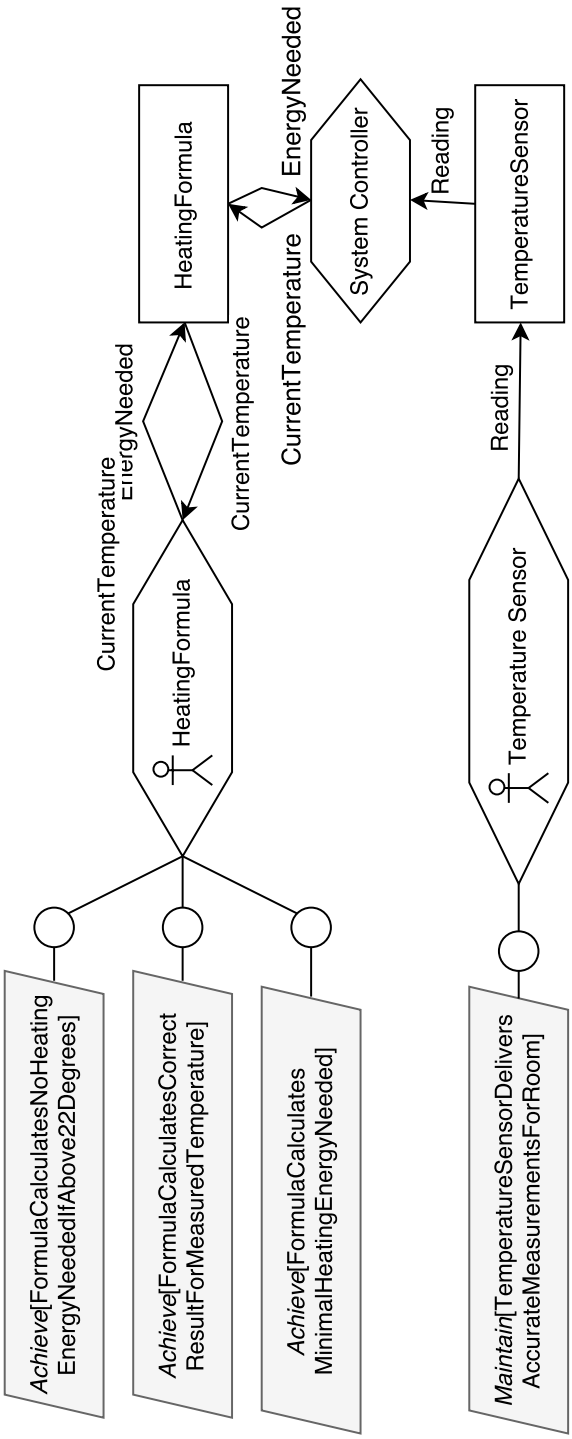

Abbildung A.9: Agent Diagram - Teil 2

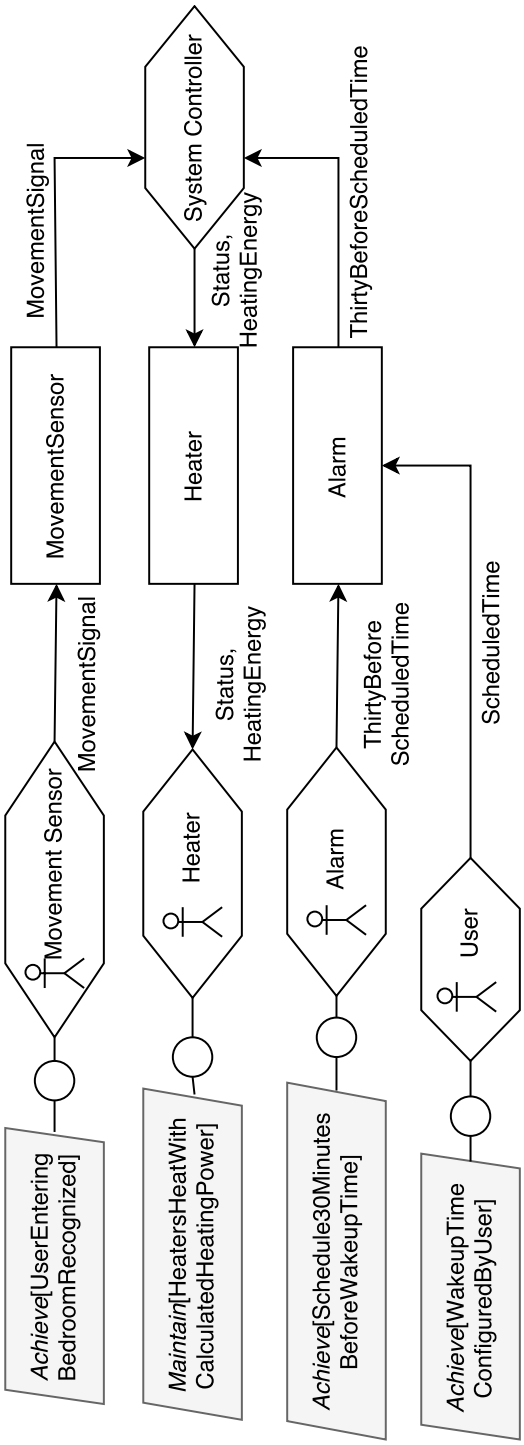

Abbildung A.10: Agent Diagram - Teil 3

A.4 Operation Model

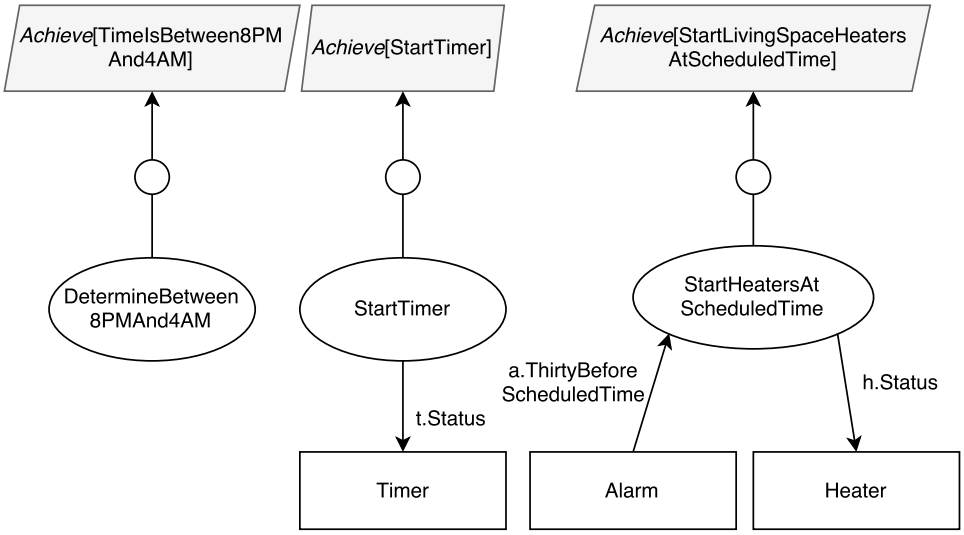

Abbildung A.11: Operationalization Diagram - *System Controller* - Teil 1

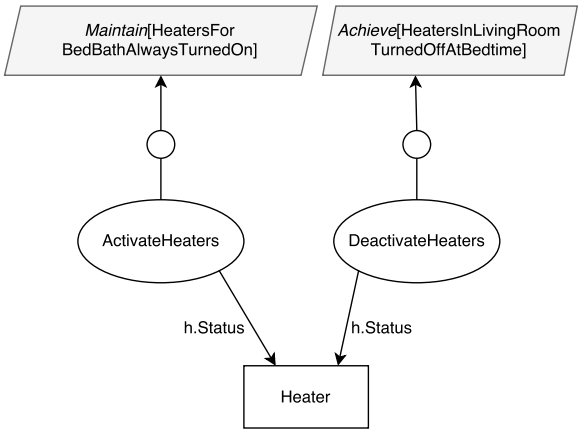

Abbildung A.12: Operationalization Diagram - *System Controller* - Teil 2

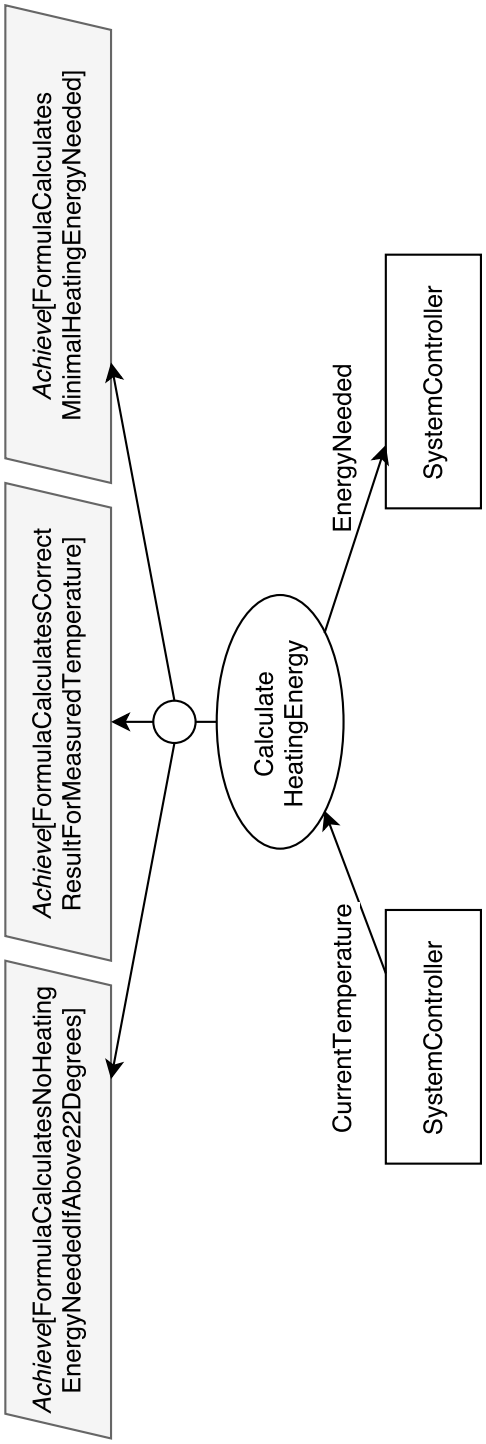

Abbildung A.13: Operationalization Diagram - *HeatingFormula*

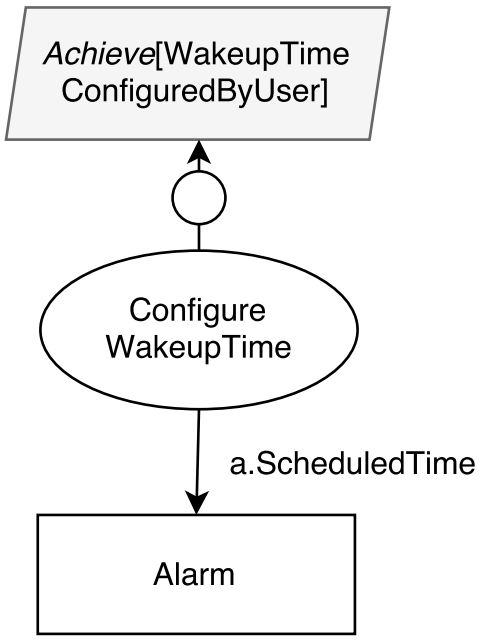

Abbildung A.14: Operationalization Diagram - *User*

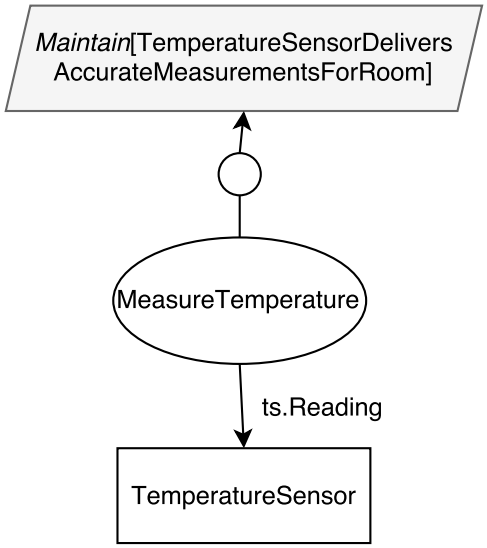

Abbildung A.15: Operationalization Diagram - *Temperature Sensor*

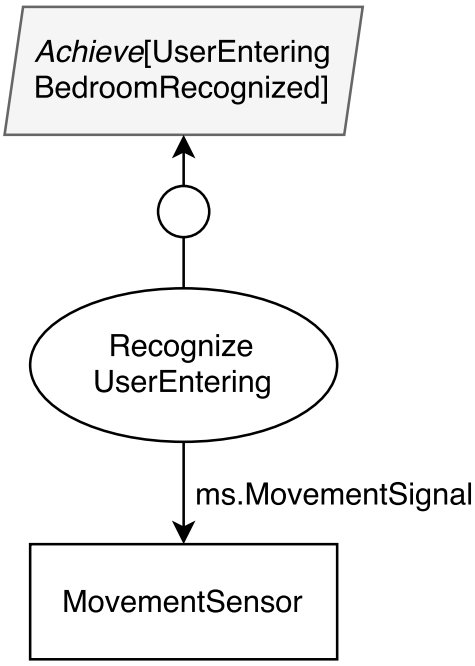

Abbildung A.16: Operationalization Diagram - *Movement Sensor*

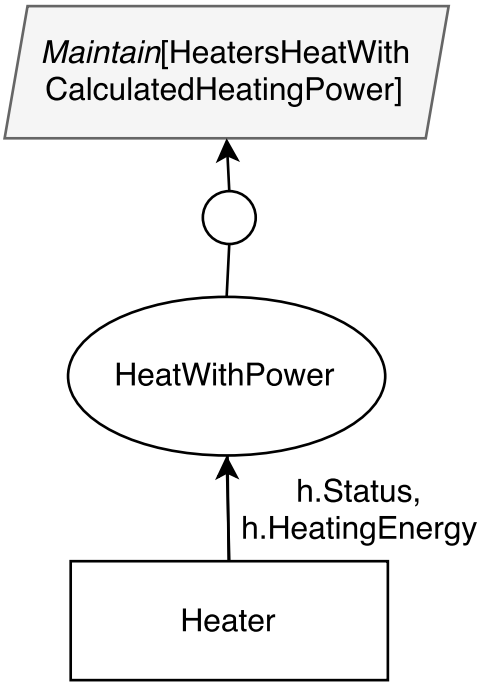

Abbildung A.17: Operationalization Diagram - *Heater*

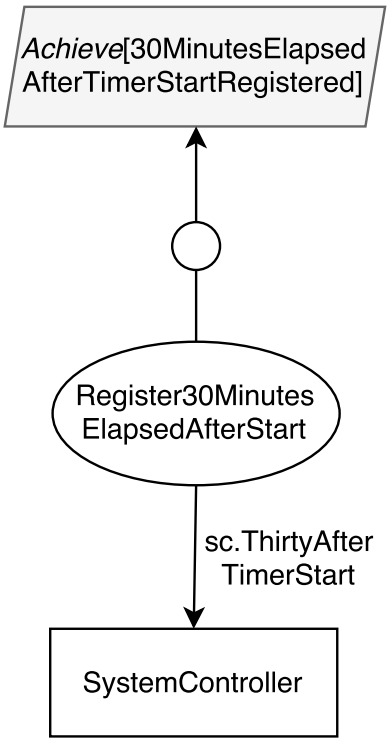

Abbildung A.18: Operationalization Diagram - *Timer*

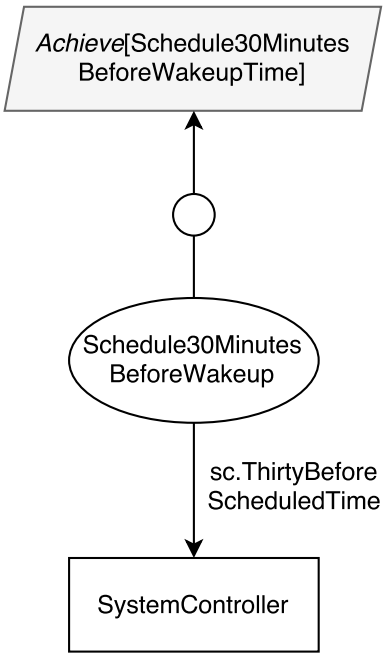

Abbildung A.19: Operationalization Diagram - *Alarm*

Anhang B

Szenariobasierte Spezifikation

B.1 Klassenmodell

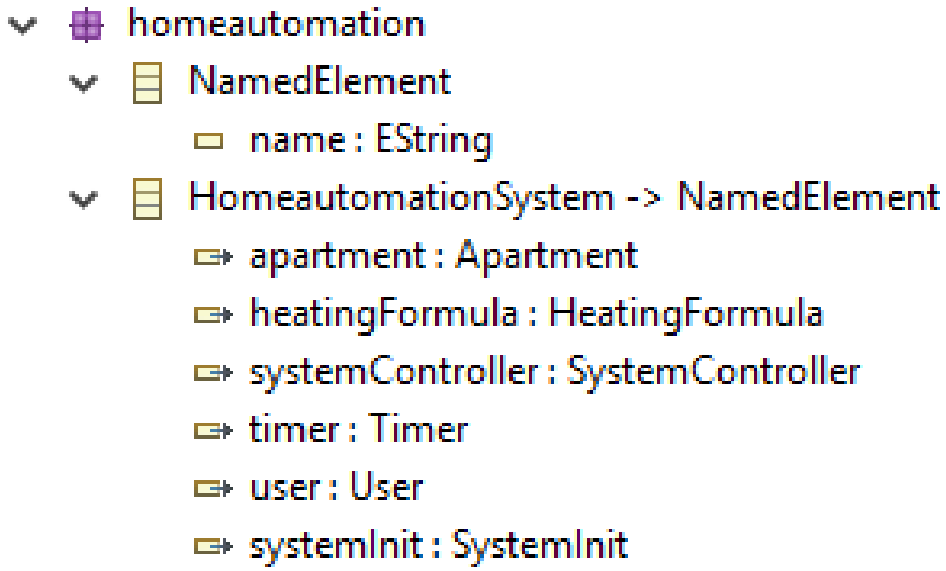

Abbildung B.1: Klassenmodell - Teil 1

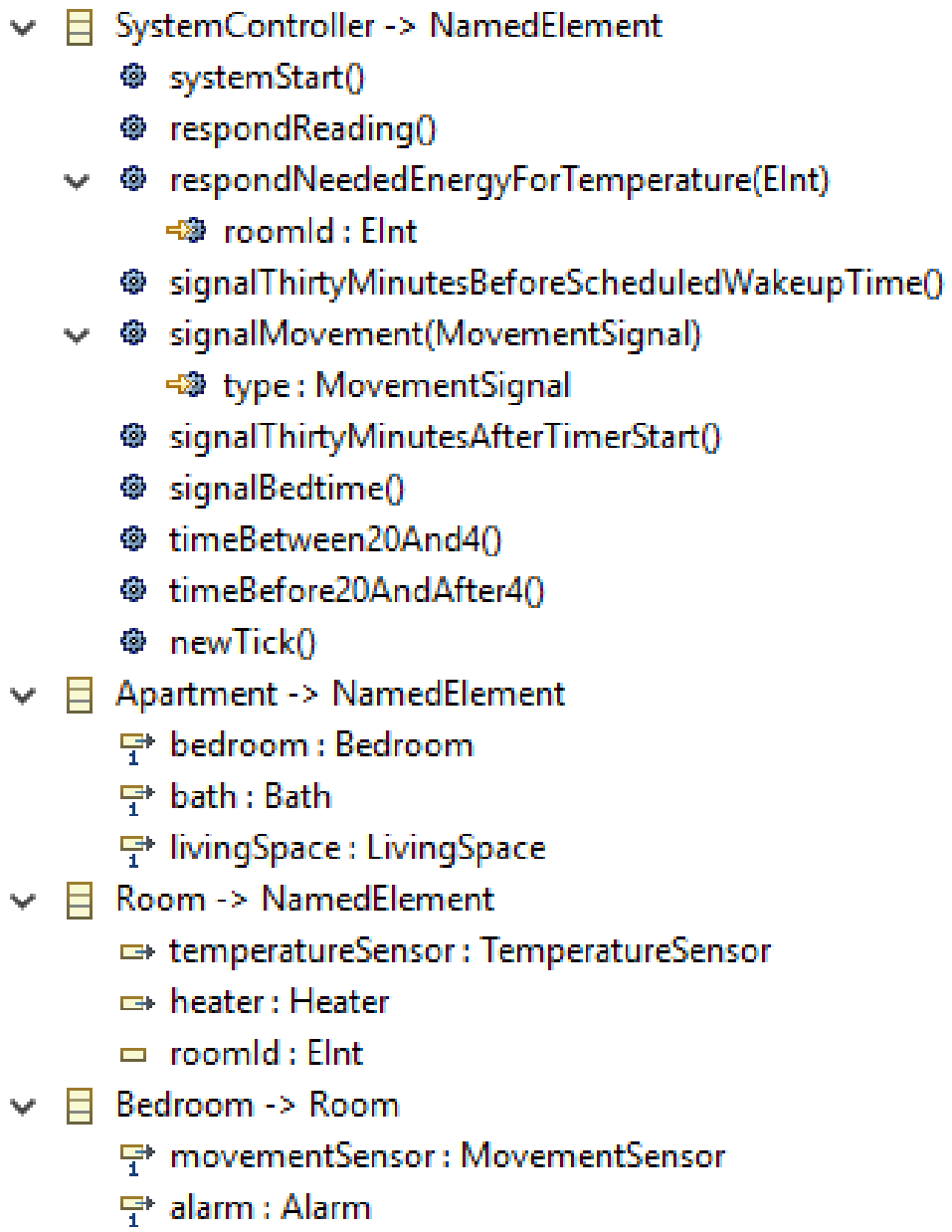

Abbildung B.2: Klassenmodell - Teil 2

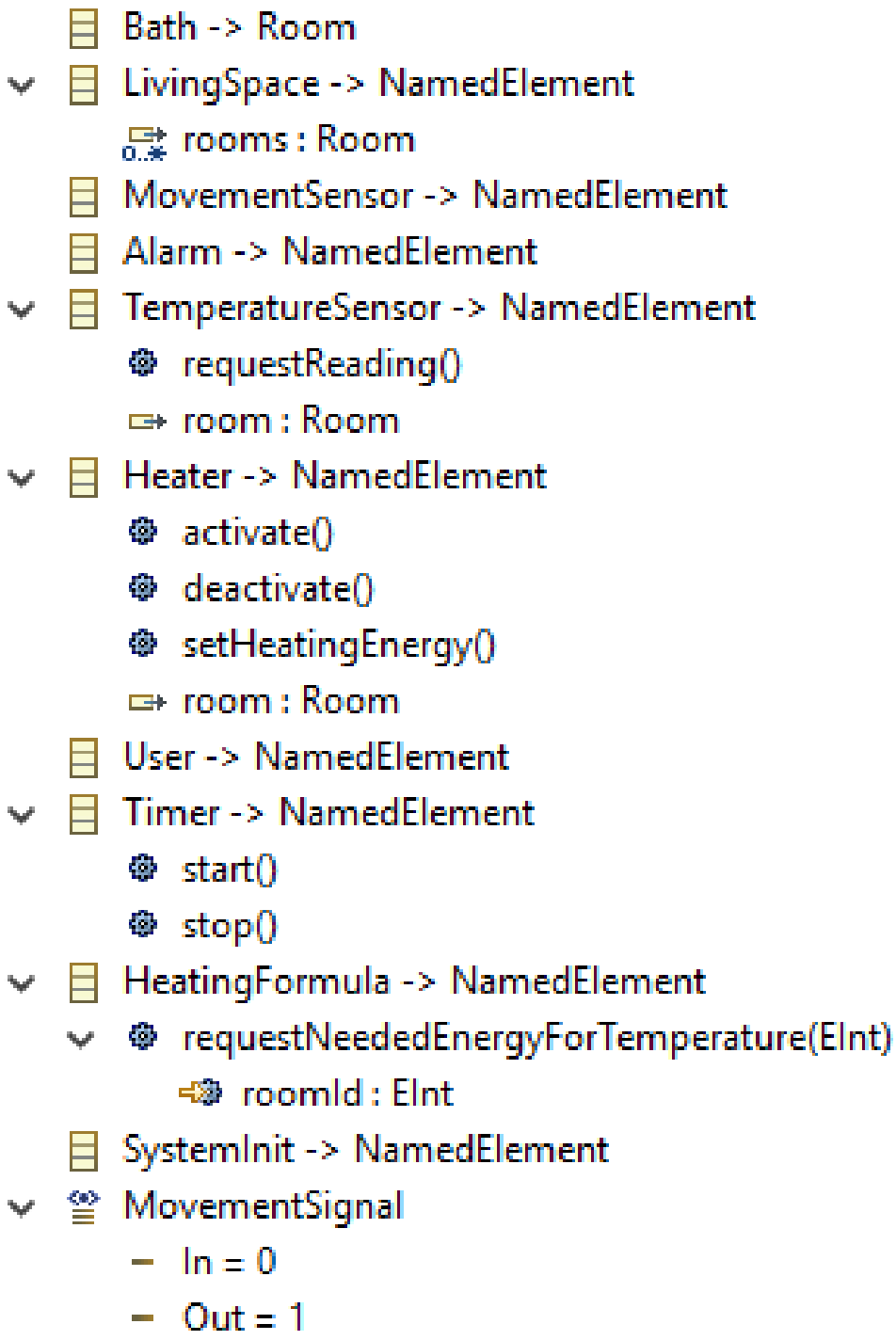

Abbildung B.3: Klassenmodell - Teil 3

B.2 Objektmodell

- Homeautomation System homeautomationSystem
  - Apartment apartment
    - Bedroom bedroom
      - Temperature Sensor bedroomTemperatureSensor
      - Heater bedroomHeater
      - Movement Sensor movementSensor
      - Alarm alarm
    - Bath bath
      - Temperature Sensor bathTemperatureSensor
      - Heater bathHeater
    - Living Space livingSpace
      - Room livingRoom
        - Temperature Sensor livingRoomTemperatureSensor
        - Heater livingRoomHeater
      - Room kitchen
        - Temperature Sensor kitchenTemperatureSensor
        - Heater kitchenHeater
    - Heating Formula heatingFormula
    - System Controller systemController
    - Timer timer
    - User user
    - System Init systemInit

Abbildung B.4: Objektmodell

B.3 SML Spezifikation

```
import "../model/homeautomation.ecore"

specification HomeAutomationSpecification {

    domain homeautomation

    controllable { SystemController }

    non-spontaneous events {
        SystemController.respondReading
        SystemController.respondNeededEnergyForTemperature
        SystemController.signalMovement
        SystemController.signalThirtyMinutesBeforeScheduledWakeupTime
        SystemController.signalThirtyMinutesAfterTimerStart
    }

    collaboration HomeAutomation {

        static role SystemController controller
        static role SystemInit systemInit
        static role Timer timer
        static role HeatingFormula heatingFormula
        static role User user

        static role Alarm alarm
        static role MovementSensor movementSensor
        static role Heater bedroomHeater

        static role Heater bathHeater

        static role LivingSpace livingSpace

        dynamic role Room room
        dynamic role TemperatureSensor temperatureSensor
        dynamic role Heater heater

        dynamic multi role Room multiRoom

        guarantee scenario BedroomAndBathHeatersActivatedOnSystemStart {
            systemInit -> controller.systemStart()
            strict committed controller -> bedroomHeater.activate()
            strict committed controller -> bathHeater.activate()
        }
```

Listing 3: SML Spezifikation - Teil 1

```
1
2           assumption scenario SystemInitialization {
3           systemInit -> controller.systemStart()
4           while {
5               alternative {
6                   movementSensor -> controller.signalMovement(MovementSignal:In
                        )
7               } or {
8                   movementSensor -> controller.signalMovement(MovementSignal:
                        Out)
9               } or {
10                  alarm -> controller.
                        signalThirtyMinutesBeforeScheduledWakeupTime()
11              }
12          }
13      } constraints [
14          forbidden systemInit -> controller.systemStart()
15      ]
16
17          // Should be modeled as an Assumption in the KAOS Goal Diagram
18      // guarantee scenario WakeupTimeConfiguredByUser {
19      //     user -> alarm.setWakeupTime()
20      // }
21
22      guarantee scenario HeatRoomsWithActiveHeaterTo22Degrees
23      bindings[
24          room = heater.room
25          temperatureSensor = room.temperatureSensor
26      ]{
27          controller -> heater.activate()
28          var EInt roomId = room.roomId
29          while {
30              systemInit -> controller.newTick()
31              eventually controller -> temperatureSensor.requestReading()
32              committed temperatureSensor -> controller.respondReading()
33              committed controller -> heatingFormula.
                    requestNeededEnergyForTemperature(roomId)
34              committed heatingFormula -> controller.
                    respondNeededEnergyForTemperature(roomId)
35              committed controller -> heater.setHeatingEnergy()
36          }
37      } constraints [
38          interrupt controller -> heater.deactivate()
39          ignore controller -> heatingFormula.requestNeededEnergyForTemperature
                    (*)
40          ignore heatingFormula -> controller.respondNeededEnergyForTemperature
                    (*)
41          ignore systemInit -> controller.newTick()
42      ]
43
44      assumption scenario TemperatureSensorDeliversAccurateMeasurementsForRoom
45      bindings[
46          room = temperatureSensor.room
47      ]{
48          controller -> temperatureSensor.requestReading()
49          temperatureSensor -> controller.respondReading()
50      }
51
52      assumption scenario HeatingFormulaCalculatesHeatingEnergy {
53          var EInt roomId
54          controller -> heatingFormula.requestNeededEnergyForTemperature(bind
                    roomId)
55          heatingFormula -> controller.respondNeededEnergyForTemperature(roomId
                    )
56      }
```

Listing 4: SML Spezifikation - Teil 2

```
            guarantee scenario StartHeating30MinutesBeforeWakeupTime
        bindings[
            multiRoom = livingSpace.rooms
            heater = multiRoom.heater
        ]{
            alarm -> controller.signalThirtyMinutesBeforeScheduledWakeupTime()
            urgent controller -> heater.activate()
        }

        guarantee scenario EndOfUseAtUserBedtime {
            movementSensor -> controller.signalMovement(MovementSignal:In)
            urgent controller -> timer.start()
            alternative {
                timer -> controller.signalThirtyMinutesAfterTimerStart()
                alternative {
                    urgent controller -> controller.timeBetween20And4()
                    urgent controller -> controller.signalBedtime()
                } or {
                    urgent controller -> controller.timeBefore20AndAfter4()
                    urgent controller -> timer.stop()
                }
            } or {
                movementSensor -> controller.signalMovement(MovementSignal:Out)
                urgent controller -> timer.stop()
            }
        }

        assumption scenario ThirtyMinutesElapsedAfterTimerStartRecognized {
            controller -> timer.start()
            timer -> controller.signalThirtyMinutesAfterTimerStart()
        } constraints [
            interrupt controller -> timer.stop()
        ]

        guarantee scenario HeatersInLivingRoomTurnedOffAtBedtime
        bindings[
            multiRoom = livingSpace.rooms
            heater = multiRoom.heater
        ]{
            controller -> controller.signalBedtime()
            urgent controller -> heater.deactivate()
        }
    }
}
```

Listing 5: SML Spezifikation - Teil 3

B.4 Runconfiguration

```
import "homeautomation.sml"

configure specification HomeAutomationSpecification

use instancemodel "HomeautomationSystem.xmi"

rolebindings for collaboration HomeAutomation {
    object homeautomationSystem.systemController plays role controller
    object homeautomationSystem.systemInit plays role systemInit
    object homeautomationSystem.timer plays role timer
    object homeautomationSystem.heatingFormula plays role heatingFormula
    object homeautomationSystem.user plays role user

    object homeautomationSystem.apartment.bedroom.alarm plays role alarm
    object homeautomationSystem.apartment.bedroom.movementSensor plays role
        movementSensor
    object homeautomationSystem.apartment.bedroom.bedroomHeater plays role
        bedroomHeater

    object homeautomationSystem.apartment.bath.bathHeater plays role bathHeater

    object homeautomationSystem.apartment.livingSpace plays role livingSpace
}
```

Listing 6: Runconfiguration

Literaturverzeichnis

[1] A. I. Anton. Goal-based requirements analysis. In *Requirements Engineering, 1996., Proceedings of the Second International Conference on*, pages 136–144. IEEE, 1996.

[2] P. Butcher. *Seven Concurrency Models in Seven Weeks: When Threads Unravel*. Pragmatic Bookshelf, 2014.

[3] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: the Tropos project. *Information systems*, 27(6):365–389, 2002.

[4] K. Chatterjee, W. Dvořák, M. Henzinger, and V. Loitzenbauer. Conditionally optimal algorithms for generalized büchi games. *arXiv preprint arXiv:1607.05850*, 2016.

[5] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[6] M. Glinz. On non-functional requirements. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 21–26. IEEE, 2007.

[7] J. Greenyer, D. Gritzner, T. Gutjahr, T. Duente, S. Dulle, F.-D. Deppe, N. Glade, M. Hilbich, F. Koenig, J. Luennemann, N. Prenner, K. Raetz, T. Schnelle, M. Singer, N. Tempelmeier, and R. Voges. Scenarios@run.time – distributed execution of specifications on IoT-connected robots. In S. Götz, N. Bencomo, G. Blair, and H. Song, editors, *Proceedings of the 10th International Workshop on Models@Run.Time (MRT 2015), co-located with MODELS 2015*, volume 1474 of *CEUR Workshop Proceedings*, pages 71–80. CEUR, 2015.

[8] D. Gritzner and J. Greenyer. Controller synthesis and PCL code generation from scenario-based GR(1) robot specifications. In *Proceedings of the 4th Workshop on Model-Driven Robot Software Engineering (accepted / to-appear)*, 2017.

[9] D. Gritzner, J. Greenyer, T. Gutahr, F. König, N. Glade, A. Marron, and G. Katz. ScenarioTools – a tool suite for the scenario-based modeling and analysis of reactive systems. *Science of Computer Programming*, 2017.

[10] D. Harel, S. Maoz, S. Szekely, and D. Barkan. PlayGo: Towards a comprehensive tool for scenario based programming. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 359–360. ACM, 2010.

[11] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming using LSCs and the Play-Engine*, volume 1. Springer Science & Business Media, 2003.

[12] D. Harel and R. Marelly. Specifying and executing behavioral requirements: The Play-In/Play-Out approach. *Software and Systems Modeling*, 2(2):82–107, 2003.

[13] A. Lapouchnian. Goal-oriented requirements engineering: An overview of the current research. *University of Toronto*, page 32, 2005.

[14] F. S. E. LUH. Szenariobasierte Spezifikationen. `http://www.se.uni-hannover.de/pages/de:projekte_scenarios`, 2017. [Online; accessed 20-August-2017].

[15] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on software engineering*, 18(6):483–497, 1992.

[16] A. Pattakou, C. Kalloniatis, and S. Gritzalis. Security and privacy requirements engineering methods for traditional and cloud-based systems: A review. *CLOUD COMPUTING 2017*, page 155, 2017.

[17] I. Respect. A KAOS tutorial, 2007.

[18] H. Sharp, A. Finkelstein, and G. Galal. Stakeholder identification in the requirements engineering process. In *Database and Expert Systems Applications, 1999. Proceedings. Tenth International Workshop on*, pages 387–391. Ieee, 1999.

[19] STANDISH GROUP. CHAOS Report (1994). `https://www.standishgroup.com/sample_research_files/chaos_report_1994.pdf`, 1994. [Online; Accessed 2017-07-05].

[20] A. Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262. IEEE, 2001.

[21] A. Van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*, volume 10. Chichester, UK: John Wiley & Sons, 2009.

[22] A. Van Lamsweerde and E. Letier. From object orientation to goal orientation: A paradigm shift for requirements engineering. In *Radical Innovations of Software and Systems Engineering in the Future*, pages 325–340. Springer, 2004.

[23] B. G. U. Weizmann Institute of Science. Behavioral programming. `http://www.wisdom.weizmann.ac.il/~bprogram/more.html`, 2012. [Online; Accessed 2017-08-22].

[24] E. S. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*, pages 226–235. IEEE, 1997.

[25] P. Zave. Classification of research efforts in requirements engineering. *ACM Computing Surveys (CSUR)*, 29(4):315–321, 1997.