technische universität
dortmund

Bachelorarbeit

Automatische Konfliktresolution für Szenarien in einer Smart-Home-Applikation – Evaluation theoretischer Ansätze und prototypische Implementierung auf Basis von Qivicon

Konstantin Tkachuk
November 2013

Gutachter:
Prof. Dr.-Ing. Olaf Spinczyk
Dr. Michael Ricken

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
http://ls12-www.cs.tu-dortmund.de

In Kooperation mit:
Anasoft Technology AG

Zusammenfassung

In dem sich gerade stark weiterentwickelnden Gebiet der eingebetteten Systeme Smart Home müssen noch viele Probleme gelöst werden. Eines davon sind die Konflikte, die entstehen, wenn Szenarien für die automatische Steuerung von Geräten definiert werden. Es muss nichtdeterministisches Verhalten des Systems verhindert werden, wozu ein automatisiertes Konfliktresolutionsverfahren benötigt wird.

Im Rahmen dieser Arbeit gilt es ein solches Verfahren im Kontext einer konkreten Smart Home Lösung (Qivicon[4]) zu realisieren. Zunächst soll mit Hilfe der existierenden Tools die grundlegende Szenarienverwaltung ermöglicht werden, wonach eine automatische Konfliktresolution zu verwirklichen ist. Hierfür werden drei verschiedene formale Ansätze auf Eignung untersucht und schließlich mit Hilfe eines davon prototypisch eine Lösung implementiert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Hintergrund

Im Laufe der Zeit ist deutlich geworden, dass eingebettete Systeme eine immer wichtigere Rolle im alltäglichen Leben des Menschen spielen. In nahezu jedem elektrischen Gerät findet man heutzutage eingebettete Rechnerhardware.

Ein aktuelles und sich gerade stark weiterentwickelndes Gebiet der eingebetteten Systeme ist das Smart Home. Darunter versteht man den Einsatz von Sensoren und Aktoren zur automatisierten Steuerung von elektronischen Geräten im Hause des Endnutzers. Dies verspricht unter anderem nachhaltige Energieeffizienz und ermöglicht das Monitoring von älteren Menschen. Im Allgemeinen bietet Smart Home dem Nutzer ein bisher unvorstellbares Level von Komfort, Flexibilität und Sicherheit; eine Unterstützung im alltäglichen Leben, die kaum zu unterschätzen ist.

Zurzeit arbeiten viele Unternehmen daran, Smart Home tatsächlich möglich zu machen. Eines davon ist der Konzern T-Systems, der mit vielen Partnern zusammen gerade eine Smart Home Lösung mit dem Namen *Qivicon* entwickelt. Zentraler Gegenstand von Qivicon ist die sogenannte Qivicon Homebase (auch "Box" genannt), an die alle elektronischen Geräte im Haus, die von Qivicon unterstützt werden, angeschlossen werden. Im Rahmen des Projekts soll eine vollständige Steuerung der Geräte sowohl per Tablet-PC, als auch im Allgemeinen über das Internet ermöglicht werden.

Die Firma Anasoft Technology AG, ein Partner der T-Systems im Rahmen von Qivicon, ist für die Software verantwortlich, die die Steuerung von Geräten per Computer ermöglichen wird. Dazu gehört sowohl die direkte Fernsteuerung von Geräten durch den Nutzer, als auch das Ermöglichen der Definition von Szenarien, sodass Geräte sich zu bestimmten Zeiten ohne zusätzlichen Input auf bestimmte Weisen verhalten.

1.2 Ziele der Arbeit

Bei der Definition von Szenarien kann es vorkommen, dass verschiedene Szenarien in Konflikt geraten. Sie können Geräten zur gleichen Zeit verschiedene Zustände zuweisen, was zu einem nicht deterministischen Verhalten des Systems führen würde.

Ziel dieser Arbeit ist es verschiedene Ansätze für das Lösen dieses Szenarienverwaltungsproblems zu analysieren und ein passendes Lösungskonzept zu entwickeln und zu implementieren.

1.3 Aufbau der Arbeit

Nach der Einleitung folgt eine kurze Vorstellung des Projekts Qivicon, des eigentlichen Problems und der Tools, mit Hilfe derer es gelöst werden muss. Daraufhin werden in Kapitel 3 drei verschiedene formale Ansätze im Bezug auf ihre Anwendbarkeit im Rahmen dieser Arbeit geprüft. In Kapitel 4 wird konkret betrachtet, wie die einzelnen Elemente der Anforderungen umgesetzt werden und wie genau die im vorherigen Kapitel diskutierten Ansätze zum Einsatz kommen.

Im folgenden Kapitel 5 wird die Implementierung des Entwurfs mit Hilfe von entsprechenden Diagrammen vorgestellt. Der Quell-Code ist auf der mit der Arbeit mitgelieferten CD einsehbar. Danach wird in Abschnitt 6 die Implementierung evaluiert. Schließlich folgt noch eine kurze Zusammenfassung der geleisteten Arbeit und ein Ausblick auf mögliche Weiterentwicklungen in Kapitel 7.

Es ist zu beachten, dass im Laufe der Arbeit die Begriffe “Szene” und “Szenario” als Synonyme betrachtet werden und daher die selbe Bedeutung haben.

Kapitel 2

Qivicon

2.1 Struktur

2.1.1 Kommunikation

Der Aufbau des Systems lässt sich gut anhand von Abbildung 2.1 erkennen. Jeder Nutzer hat innerhalb seiner Wohnung direkten Zugriff auf seine Qivicon Homebase. Alle angeschlossenen Geräte lassen sich per WLAN steuern, eine Internetverbindung ist nicht notwendig.

Jede Qivicon Homebase ist mit dem Qivicon-Backend-Server (QBS) verbunden. Der QBS stellt den Zugriff auf bestimmte Boxen dem Serviceportal-Backend-Server (SBS) zur Verfügung. Um auf eine konkrete Box zugreifen zu können, wird der entsprechende Username und das Passwort benötigt.

Der Nutzer kann sich per Internet auf dem SBS einloggen und Zugriff auf seine persönliche Box bekommen. Dies bietet ihm die direkte Kontrolle seines Hauses von jedem Ort mit Internetanschluss auf der gesamten Welt.

Der Serviceportal-Backend-Server bietet eine Reihe von Services an. Zentraler Aspekt darunter ist der Zugriff auf die Qivicon Homebase, aber auch Services Drittanbieter (z.B. Pizza, Taxi, usw.) sind enthalten.

2.1.2 Qivicon Homebase

Die Qivicon Homebase ist ein Home Automation Gateway, dass in einer Wohnung installiert wird und die Kontrolle über elektronische Geräte im Haushalt bietet. Es benötigt lediglich einen Anschluss an das Internet und eine Kopplung zum entsprechenden Gerät. Die Kommunikation zwischen Gerät und Box findet über Funk statt.

Qivicon hat eine serviceorientierte Architektur, basierend auf dem Java OSGi Framework[10]. OSGi spezifiziert eine dynamische Softwareplattform, die hardwareunabhängig ist und es

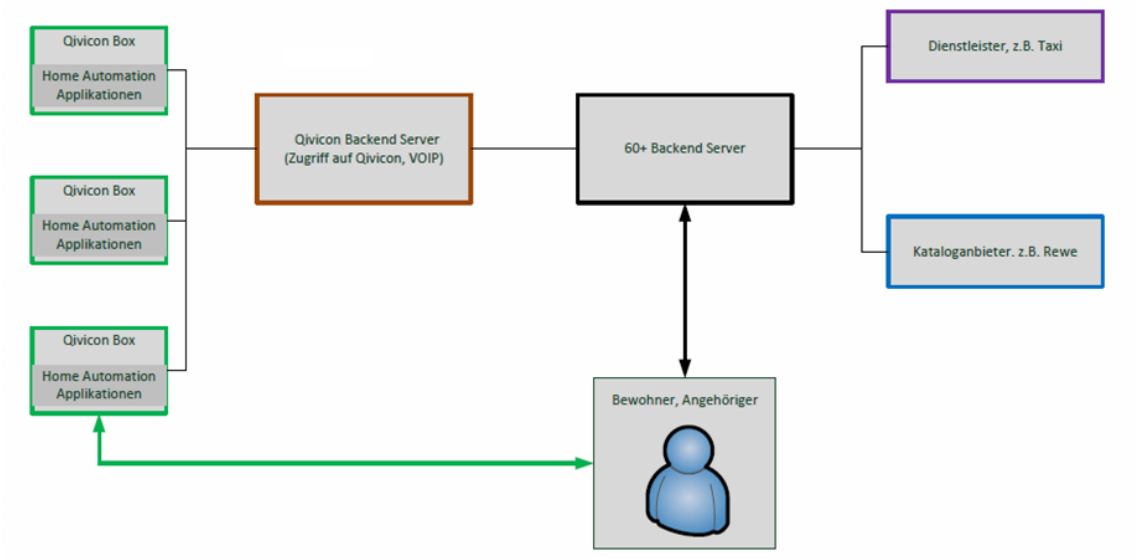

Abbildung 2.1: Aufbau der Kommunikation zwischen Box, Server und Client

ermöglicht, Anwendungen zu modularisieren und zu verwalten. Der Funktionsweise der Kommunikation ist in Abbildung 2.2 veranschaulicht.

Im Rahmen des OSGi Frameworks wird das gesamte Programm in verschiedene Softwarekomponenten ("Bundles") aufgeteilt. Jede Komponente bietet und bezieht Services. Die angebotenen Services werden von den entsprechenden Bundles zur Laufzeit im System registriert, wonach andere Bundles, die diese Services beziehen, darüber informiert und ihrerseits gestartet werden können. Ein Bundle wird erst gestartet, wenn alle von ihm benötigten Services im System registriert wurden.

Ein derartiger modularer Aufbau erlaubt es verschiedene Bundles nahezu unabhängig von einander zu entwickeln und später einzelne Bundles gegen aktuellere Versionen problemlos auszutauschen. Aus einer Reihe solcher untereinander kommunizierender Softwarekomponenten besteht auch die Anwendung *Serviceportal*, an der die Anasoft Technology AG gerade arbeitet.

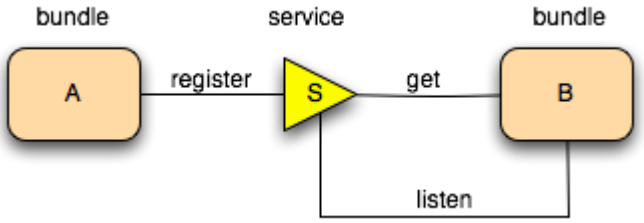

Abbildung 2.2: OSGi Komponentenmodell: Kommunikation zwischen Bundles[7]

2.1.3 Serviceportal

Serviceportal ist eine Anwendung, die eine Reihe von Services anbietet. Dazu gehören Gerätesteuerung, Szenarioverwaltung, Hausüberwachung und verschiedenste Services Drittanbieter. In erster Linie ist sie jedoch an ältere Menschen (60+) gerichtet. Das spiegelt sich vor allem auf der graphischen Benutzer-Schnittstelle (GUI) wieder.

Die GUI ist eine Webanwendung, die hauptsächlich durch Javascript, basierend auf der Bibliothek *backbone.js*[2], erzeugt wird. Das führt dazu, dass sie sich auch auf mobilen Geräten problemlos ausführen lässt.

2.2 Szenarien

2.2.1 Einleitung

Die Automatisierung der Haushaltsgeräte soll vor allem durch Szenarien realisiert werden. Unter einem Szenario versteht man die Zuordnung von bestimmten Zuständen bestimmter Geräte an bestimmte Zeiträume. Szenarien sollen entweder sich in festgelegten Zeiträumen regelmäßig wiederholen oder manuell ausführbar sein.

Beispiele

1. Alle Lichter im Wohnzimmer sollen um 18-20 Uhr an sein.
2. Die Jalousien sollen regelmäßig um 18 Uhr runter gehen.
3. Alle Lichter im Haus sollen ausgeschaltet werden.

2.2.2 Gegeben

Das Qivicon Home Automation Management (HAM)[5] ist eine Regelmaschine und bietet eine Reihe von Tools, um derartige Szenarien zu definieren. Es existieren Conditions, Commands, Scenes und Rules.

Conditions

Conditions sind Bedingungen, nach deren Erfüllung Events geworfen werden. Aktuell gibt es verschiedene zeit-basierte Konditionen, wie z.B. die *AbsoluteTimerCondition* und die *PeriodicTimerCondition*. Der Absolute Timer publiziert einmal ein Event, wenn der vorher eingestellte Moment eintrifft. Ein Periodic Timer publiziert solche Events immer wieder in einem bestimmten Zeitabstand.

Es gibt noch eine Reihe weiterer zeit-basierter Bedingungen. Außerdem lassen sich auch *Custom Conditions* definieren, wo der Entwickler beliebige Bedingungen selbst einprogrammieren kann.

Commands

Ein *Command* ist eine Aktion, z.B. das Überführen eines Gerätes in einen bestimmten Zustand oder das Aufrufen einer durch das Interface des Gerätes definierter Methode. Wenn ein Command ausgelöst wird, wird die entsprechende Aktion ausgeführt.

Auch hier lassen sich durch den Entwickler beliebige *Custom Commands* definieren.

Scenes

Eine *Scene* ist eine besondere Art von *Command*. Sie ist eine Ansammlung von beliebig vielen anderen Commands, das heißt, bei dem Ausführen einer Scene werden alle in ihr enthaltenen Befehle ausgeführt. Eine solche Zusammenfassung vereinfacht deutlich die Definition von Rules.

Rules

Ein Rule verbindet *eine* Condition und *ein* Command. Es ist eine Regel, die besagt, dass sobald eine bestimmte Condition ein Event publiziert, ein bestimmtes Command ausgeführt wird. Hier helfen die Scenes mehrere Commands an eine Condition zu binden, ohne einzelne Rules für jedes Command definieren zu müssen.

2.2.3 Ansatz

Konzept

Insgesamt lässt sich aus den oben beschriebenen Elementen bereits eine primitive Gerätesteuerung realisieren. Es wird eine Scene definiert, die mit Hilfe eines Rule an eine Condition gebunden wird. Damit lassen sich beliebige Geräte zu einer bestimmten Zeit auf einen bestimmten Zustand schalten.
Leider führt eine derartige Realisierung sofort zu einer ganzen Reihe von Problemen.

Probleme

Bei einer direkten Implementierung mit Hilfe der bereits vorhandenen Tools stößt man sofort auf eine ganze Reihe von Problemen.

Was passiert, wenn

- zwei sich widersprechende Commands an ein Event gebunden werden?
  Beispiel: Ein Command schaltet das Gerät ein, das andere schaltet es aus. Welchen Zustand wird das Gerät nach Ausführung des Rule haben?
- ein Szenario über einen bestimmten Zeitraum aktiv sein soll?
  Beispiel: Ein Gerät soll über einen bestimmten Zeitraum an sein. Zurzeit gibt es keine Möglichkeit Zustände für Zeiträume zu definieren.

- zwei solche Szenarien sich widersprechen?
  Beispiel: Zwei Szenarien definieren für ein Gerät verschiedene Zustände im selben Zeitraum. Welchen Zusand wird das Gerät nach dem Schalten haben?
- ein Nutzer ein Gerät manuell schaltet? Wie reagiert ein aktives Szenario?
  Beispiel: Ein Szenario definiert, das ein Gerät noch eine Stunde an sein soll. Nun schaltet der Nutzer das Gerät manuell aus. Wie soll das System reagieren?
- man einen Minimalwert für ein Gerät definieren möchte? Lässt sich das Gerät dann manuell nicht ausschalten?
  Beispiel: Man möchte ein Nachtlicht-Szenario definieren, sodass das Licht nicht unter 10% Stärke leuchten darf, damit ältere Menschen sehen, wohin sie gehen. Beim manuellen Schalten kann der Nutzer das Licht komplett einschalten, aber nicht komplett ausschalten. Das führt jedoch auch dazu, dass bei einer Definition eines Szenarios mit Minimalwert 100% das Licht auch manuell nicht ausschaltbar ist.
- ein Szenario zu Ende ist?
  Beispiel: Das Szenario ist zu Ende. Soll der vorherige Zustand wiederhergestellt werden? Soll nichts getan werden?

Zurzeit ist das Verhalten des Systems nicht deterministisch, das heißt es lässt sich nicht im Voraus vorhersagen, welches Event zuerst bearbeitet wird, wenn zwei Conditions zur gleichen Zeit Events publizieren. Analog, wenn zwei sich widersprechende Commands an eine Condition gebunden werden.

2.2.4 Anforderungen

Als Erstes müssen gewisse Entscheidungen auf der Anforderungsebene getroffen werden. Nach detaillierter Analyse der vorhandenen Tools und der Probleme, die gelöst werden müssen, wurden bei Anasoft Technology AG folgende Entscheidungen getroffen:

1. Szenen werden Prioritäten erhalten.
   Bei einem Konflikt zwischen zwei Szenarien wird das Szenario mit der höheren Priorität ausgeführt.

2. Es gibt keinen Endzustand.
   Es wird vor dem Ausführen eines Szenarios nicht gespeichert, welche Zustände die Geräte gerade haben. Entsprechend werden auch keine Zustände am Ende des Szenarios definiert. Innerhalb des Szenarios wird auch kein Endzustand definiert.

3. Am Ende eines Szenarios werden die anderen aktiven Szenarien nochmal ausgeführt, mit Priorität von hoch bis niedrig.
   Falls es keine aktiven Szenarien gibt, so gibt es keine Reaktion seitens des Systems.

Es sollte für alle Geräte Szenarien für 24 Stunden täglich geben, damit 3. ideal funktioniert.

4. Manuelle Steuerung übersteuert aktive Szenarien.
   Falls ein Mensch ein Gerät manuell schaltet, so nimmt das Gerät den entsprechenden Zustand an. Eine Ausnahme sind nur Minimal-Zustände.

5. Nach manueller Schaltung wird eine Sperrzeit t eingestellt, während der Geräte nicht durch Szenarien geregelt werden können.
   Dies ist gemacht, damit z.B. nicht das Licht ausgeht, weil eine Szene es so definiert, während ein Mensch unter der Dusche steht.

6. Es gibt Geräte, für die Minimalzustände definiert werden können.
   Wie bei dem Nachtlicht-Problem erläutert, kann man für Gerät einen Minimalwert definieren, sodass z.B. ein Gerät nicht ausgeschaltet werden kann.

Ziel der Arbeit ist es diese Anforderungen im Rahmen von Serviceportal umzusetzen, also eine Lösung zu implementieren. Zunächst muss das Problem jedoch noch formalisiert werden.

Kapitel 3

Ansätze

Bevor verschiedene Ansätze evaluiert werden können, muss erst klar gemacht werden, welche Daten zur Verfügung stehen und was das Ziel ist.

Ein Szenario lässt sich schematisch gut als Tupel $< T, G, P >$ darstellen. T steht für die Zeit, bzw. das Zeitintervall, wann das Szenario ausgeführt werden soll, G identifiziert den Zielzustand und P ist die Priorität in einem Konfliktfall.

Es besteht ein Konflikt zwischen zwei Szenarien, wenn sie sich sowohl in T, als auch in G überschneiden. Ziel ist es eine Menge von Szenarien auf solche Konflikte zu prüfen und dafür zu sorgen, dass in jedem Konfliktfall die betroffenen Szenarien verschiedene Prioritäten haben.

Die Prioritätenvergabe in einem Konfliktfall kann einerseits sehr trivial ausfallen, z.B. durch Anwendung des Zufallsprinzips, andererseits beliebig komplex werden, wenn der Nutzer eigene Relationen zwischen Szenen definiert. Es könnte sich dabei um eine simple binäre Relation, wie "eine Szene hat höhere Priorität als eine andere" handeln, aber auch um z.B. "eine Szene, die eine sowohl Mitteilungsbenachrichtigung, als auch eine Kontaktänderung realisiert, hat höhere Priorität, als eine Anwesenheit-Simulation-Szene".

Ziel ist es ein Lösungskonzept für die Konfliktfindung und die Konfliktresolution zu entwickeln. Im Folgenden werden drei verschiedene Ansätze analysiert.

3.1 Temporal Logic of Actions

3.1.1 Allgemein

Ein möglicher Ansatz zum Entwickeln eines Lösungskonzepts ist es eine formale Spezifikation zu erstellen. Es ist ein Algorithmus gesucht, der eine beliebige Anzahl von Eingaben (Szenarien) nach bestimmten Kriterien untersucht und berechnet, ob sich einige davon widersprechen. Falls dies funktioniert, wäre es eventuell möglich von dem Ergebnis aus zurück zu schließen, *welche* Szenarien in Konflikt stehen.

Eine mögliches Framework zur Implementierung dieses Ansatzes ist TLA+[13], was aus der „Temporal Logic of Actions“ Spezifikationssprache und der PlusCal[14] Algorithmen-Sprache besteht. Dabei werden in PlusCal geschriebene Algorithmen automatisch in TLA Algorithmen umgewandelt. TLA+ ist Open Source, was es für diese Arbeit interessant macht.

Als Algorithmen-Sprache unterscheidet sich PlusCal in vielen Hinsichten von gewöhnlichen Programmiersprachen. Im Wesentlichen lassen sich drei Unterschiede hervorheben[14]:

1. Eine Algorithmen-Sprache operiert auf beliebigen komplexen mathematischen Objekten. Eine Programmiersprache hingegen arbeitet mit simplen Objekten, wie **boolean** und **int**, aus denen die komplexen Objekte zusammengesetzt werden müssen.

2. Ein Algorithmus besteht aus einer Sequenz von atomaren Operationen. Es gibt keine feste Definition einer atomaren Operation in einem Programm.

3. Ein Algorithmus beschreibt eine Klasse von möglichen Berechnungen, während ein Programm nur eine bestimmte davon realisiert.

Was die tatsächliche Funktionsweise betrifft, in TLA+ werden ein Initial- und ein Folgezustand mit Hilfe der temporalen Logik der Aktionen[13][6] definiert. Danach läuft der Model-Checker automatisch solange alle möglichen Schritte durch, bis entweder ein Deadlock entsteht oder alle möglichen Zustände berechnet wurden. Es gibt natürlich den Sonderfall, wo der Algorithmus nicht terminiert.

Temporale Logik wird benutzt, um die Initial- und Folgezustände zu definieren. Außerdem lassen sich beliebige Invarianten für den Algorithmus definieren, um zu prüfen, ob auch tatsächlich alle Einschränkungen eingehalten werden.

Es ist sinnvoll TLA+ einzusetzen, um ein bestimmtes Lösungskonzept auf allgemeine Korrektheit zu prüfen. Die Erstellung einer TLA+ Spezifikation erlaubt es formal zu detaillieren, *was* berechnet werden muss, ohne auf die konkrete Implementierung einzugehen.

3.1.2 Im Bezug zu Szenarien

Bezüglich der Verwendung von TLA+ zur Lösung des Szenenverwaltungsproblems, stellt sich die Frage, welche Vorteile ein solcher Einsatz mit sich ziehen würde. In Sektion 2.2.4 wurde bereits eine informelle Spezifikation des Problems vorgestellt. Es wurde aufgelistet zu was das Programm in der Zukunft fähig sein muss. Ein Algorithmus zur Prüfung und Lösung der Konflikte zwischen Szenarien fällt relativ trivial aus:

1. Prüfe, ob die Zeitintervalle von jeweils zwei Szenarien sich überschneiden.

2. Prüfe, ob die gesteuerten Geräte von zwei Szenarien sich überschneiden.

3. Prüfe, ob die Prioritäten verschieden sind.

4. Falls die Prioritäten gleich sind, vergib neue Prioritäten.

Man könnte die Szenen, wie am Anfang von Kapitel 3 vorgestellt, als Tupel darstellen. Ferner lassen sich die ersten drei Punkte des oben gezeigten Algorithmus leicht mit PlusCal implementieren. Allerdings werden dadurch weder neue Erkenntnisse gewonnen, mit Ausnahme der absoluten Sicherheit, dass der Algorithmus korrekt ist, noch hilft es bei der Berechnung von neuen Prioritäten.

Die Frage bleibt offen, *wie* man neue Prioritäten vergeben soll. Im Rahmen der Spezifikation könnte man zwar das Zufallsprinzip verwenden, aber für das endgültige Lösungskonzept wäre es inakzeptabel. Es bleibt unklar, wie genau die komplexen Relationen zwischen Szenarien berücksichtigt werden sollen. Die Entwicklung eines allgemeinen Algorithmus für die Konfliktresolution scheint sehr komplex auszufallen, besonders da es keinen genauen Ansatzpunkt gibt.

Zusätzlich gibt es derzeit keine Integration von Java mit TLA+, was dessen direkte Verwendung zwar nicht unmöglich macht, aber durchaus erschwert.

3.1.3 Fazit

Obwohl der Einsatz von TLA+ zur Erstellung einer formalen Spezifikation des Problems möglich ist, würde sie keine besonderen Vorteile bei der Entwicklung des konkreten Lösungskonzepts bieten. Man könnte beweisen, dass für alle theoretisch erstellbaren Szenarien die Konflikte *auffindbar* und *lösbar* sind, aber wäre keinen Schritt weiter in der tatsächlichen Konfliktresolution.

3.2 Business Rules Management System

3.2.1 Allgemein

Moderne komplexe Geschäftssysteme sind gewöhnlich in mehrere Schichten aufgeteilt: *Präsentation*, *Business Logic* und *Persistierung*. Die Schicht *Business Logic* stellt dabei den Kern der Applikation dar; hier sind alle Prozesse und Entscheidungen widergespiegelt.

Leider ändern sich mit der Zeit die Anforderungen an diese Schicht, was zu einem hohen Maß an Wartungsarbeiten führt. Lösungen mit Hilfe von gewöhnlichen Programmiersprachen, wie *Java* und *C#*, würden schnell zu sogenanntem *Spaghetti*-Code[8] führen, einer unzähligen Menge von *if/else*-Ausdrücken, die unübersichtlich und unflexibel sind.

Als Lösung dieses Problems wurden die *Business Rules Management Systems* (BRMS) entwickelt. Die Idee dahinter ist es die Business Logic an einen externen Ort in Form von *Rules* (Condition/Command-Paare) auszulagern, um die Realisierung von der Logik zu entkoppeln und damit ein hohes Maß an Flexibilität und und Wiederverwendbarkeit zu erreichen.

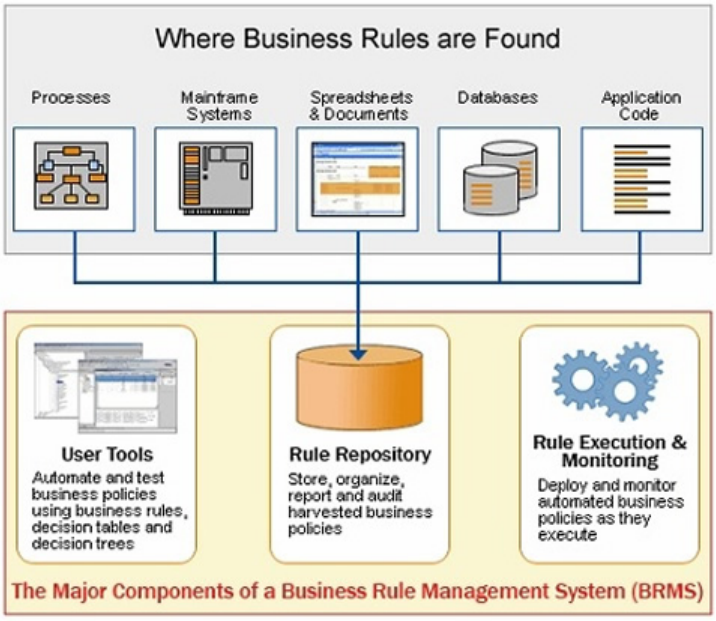

Abbildung 3.1: Aufbau eines BRMS[1]

Ein BRMS besteht aus drei Teilen: *Editor*, *Repository* und *Execution Core*. Der *Editor* bietet ein User Interface, in dem sich Rules leicht erstellen lassen, auch durch Domänen-Experten. In der *Repository* werden die Rules gespeichert und durch den *Execution Core* ausgeführt. In Abbildung 3.1 ist der Aufbau veranschaulicht.

Ein solcher deklarativer Aufbau bietet den Entwicklern eine Reihe von Vorteilen[8], vor allem Flexibilität, Wiederverwendbarkeit und Übersichtlichkeit. Gerade bei komplexen Geschäftssystemen spielt auch der Fakt, dass Domänen-Experten Rules definieren können, ohne sich in die tatsächliche Realisierung einarbeiten zu müssen, eine wichtige Rolle.

Es ist sinnvoll BRMS einzusetzen, wenn eine Reihe von Anforderungen[8] erfüllt sind. Falls in der zu entwickelnden Applikation ein hohes Maß an sich ständig verändernden komplexen Entscheidungen getroffen werden, so kann der Einsatz eines BRMS empfohlen werden. Es heißt, dass erst nach mindestens einem Jahr Nutzung der investierte Aufwand sich lohnt[16].

3.2.2 Im Bezug zu Szenarien

Die Szenarienverwaltung lässt sich mit Sicherheit auch mit BRMS implementieren. Schließlich handelt es sich hierbei nur um einen anderen Stil der Programmierung, bei dem die Logik, die gewöhnlich in verschiedenen *Controllern* angesiedelt ist, an einen externen Ort verlagert wird. Das bedeutet, dass BRMS alles kann, zu was ein gewöhnliches Java-Programm auch in der Lage ist.

Entsprechend könnten auch BRMS in der Szenensteuerung Einsatz finden. Man könnte alle Szenarien (*Fakten*) zu der Laufzeitumgebung (*Session*), in der die Rules evaluiert

werden, hinzufügen, wo die Rules sie auf Konflikte prüfen würden. Ferner könnte man mit Rules sogar eine allgemeine, wenn auch simple, Resolutionsmethode für die Konfliktfälle definieren. Eine Validierung von Szenen wäre auch durchaus realisierbar.

Leider tritt auch hier ein Problem bei der Berücksichtigung manuell erstellter Relationen auf. Um besonderes Verhalten für einzelne Szenen zu definieren, müsste man jedes mal zusätzliche, nur die Szene in Frage betreffenden Rules definieren, was auf Dauer zu einem immer wachsenden Overhead führen würde. Außerdem bleibt es unklar, *wie* genau die komplexe Prioritätenvergabe realisiert werden soll.

Darüber hinaus könnte die Tatsache, dass eine aktivierte Rule nicht direkt reversibel ist, also dass eine durch eine Ruleausführung hervorgerufene Änderung nicht direkt umgekehrt werden kann, zu Problemen führen. Eine derartige Funktionalität könnte mit Sicherheit zusätzlich implementiert werden, allerdings würde es den Einsatz von z.B. verschiedenen Backtracking-Algorithmen deutlich erschweren.

Die Frage, die daher beantwortet werden muss, ist nicht ob man BRMS benutzen *kann*, sondern ob man es *soll*. Die Form, in der Rules in der Szenarienverwaltung eingesetzt werden können, ist eigentlich nicht dafür gedacht. Ziel war es die allgemeine Logik des Programms auszulagern um Flexibilität zu gewinnen, nicht unzählige Rules pro hinzukommende Szene zu definieren.

Des weiteren, handelt es sich bei der Szenarienverwaltung eher um ein statisches Problem. Es ist unwahrscheinlich, dass im Laufe der Zeit sich *viele* Implementierungsdetails ändern werden. Außerdem gibt es keine externen Domänen-Experten, deren Input wichtig wäre.

Da im Grunde keine der Voraussetzungen für den *empfohlenen* Einsatz von BRMS erfüllt ist, ist von ihrer Verwendung abzuraten.

3.2.3 Fazit

Das Problem der Szenarienverwaltung ist zwar in sich relativ komplex, aber auf Dauer eher statisch. Nachdem eine Lösung einmal fertig ist, ist es unwahrscheinlich, dass sich viel an ihr ändern wird. Das führt dazu, dass der Einsatz von BRMS, obwohl möglich, sich wahrscheinlich nicht lohnen wird, da einerseits der zusätzliche Aufwand zur erstmaligen Implementierung deutlich höher wird, andererseits die Performanz einer solchen Implementierung geringer ist, als einer gewöhnlichen, imperativen[8] Lösung.

3.3 Constraint Satisfaction Problem

3.3.1 Allgemein

Allgemein betrachtet ist ein Constraint Satisfaction Problem (CSP) ein Problem, bei dem man eine Reihe von Variablen hat, denen man gewisse Werte zuweisen möchte. Jede Va-

riable hat eine Domain, das heißt einen Wertebereich, in dem ein passender Wert gefunden werden soll. Außerdem gibt es pro Variable Einschränkungen, (z.B. dass ihr Wert nicht gleich dem Wert einer anderen Variablen sein soll). Ein CSP gilt als gelöst, wenn eine Variablenbelegung gefunden wurde, die alle Einschränkungen erfüllt.

Formaler betrachtet wird ein Constraint Satisfaction Problem durch ein Tripel (V, D, C) beschrieben, wobei $V = \{v_1, ..., v_n\}$ eine endliche Menge von Variablen mit assoziierten Wertebereichen $D = \{D_1, ..., D_n\}$ mit $\{v_1 : D_1, ..., v_n : D_n\}$ ist. C ist eine endliche Menge von Constraints $C_j(V_j)$, $j \in \{1, ..., m\}$, wobei jedes Constraint $C_j(V_j)$ eine Teilmenge $V_j = \{v_{j1}, ..., v_{jk}\} \subseteq V$ der Variablen zueinander in Relation setzt und deren gültige Wertekombinationen auf eine Teilmenge von $D_{j1} \times ... \times D_{jk}$ beschränkt[17].

Constraint Satisfaction Probleme sind oft NP-vollständig, was zu entsprechenden worst-case Laufzeiten führen kann. Sie werden in der Regel durch verschiedene Suchalgorithmen gelöst. Allgemein betrachtet sind alle Lösungsverfahren abgeleitet von zwei grundlegenden Ideen:

1. Backtracking

Beim Backtracking geht es um die systematische Abarbeitung aller möglichen Kombinationen von Variablenbelegungen. Dabei werden den Variablen nacheinander zufällige gültige Werte zugewiesen, solange die Constraints es erlauben. Falls es keine gültige Variablenbelegung gibt, so wird der vorherigen Variable ein anderer gültiger Wert zugewiesen und man versucht es nochmal. Dieser Vorgang wird wiederholt, bis entweder alle Permutationen geprüft wurden, oder bis eine Lösung gefunden wurde.

In Abbildung 3.2 ist eine durch Nutzung von Backtracking resultierende Baumstruktur für das Damenproblem dargestellt. Dabei geht es darum, n Damen auf einem $n \times n$ Schachbrett so aufzustellen, dass keine der Damen eine andere schlagen kann.

Wie schematisch gezeigt ist, versucht der Algorithmus die vier Damen nacheinander auf die vier Vertikalen zu verteilen. Dabei fängt er mit einer Dame in der linken oberen Ecke an und versucht dann für die nächste Dame einen gültigen Platz auf der nächsten Vertikale zu finden. Wenn bemerkt wird, dass es keinen gültigen Platz auf einer Vertikale gibt, so findet das eigentliche Backtracking statt und die vorherige Variable wird verändert.

2. Constraint Propagation

Unter Constraint Propagation versteht man die Analyse der existierenden Constraints und die Berechnung daraus neuer Einschränkungen. Ziel ist es dabei entweder sofort einen Widerspruch auf Constraint-Ebene festzustellen, der das Problem nicht lösbar machen würde, oder neue Constraints zu definieren (z.B. Wertebereiche weiter eingrenzen), sodass bei der

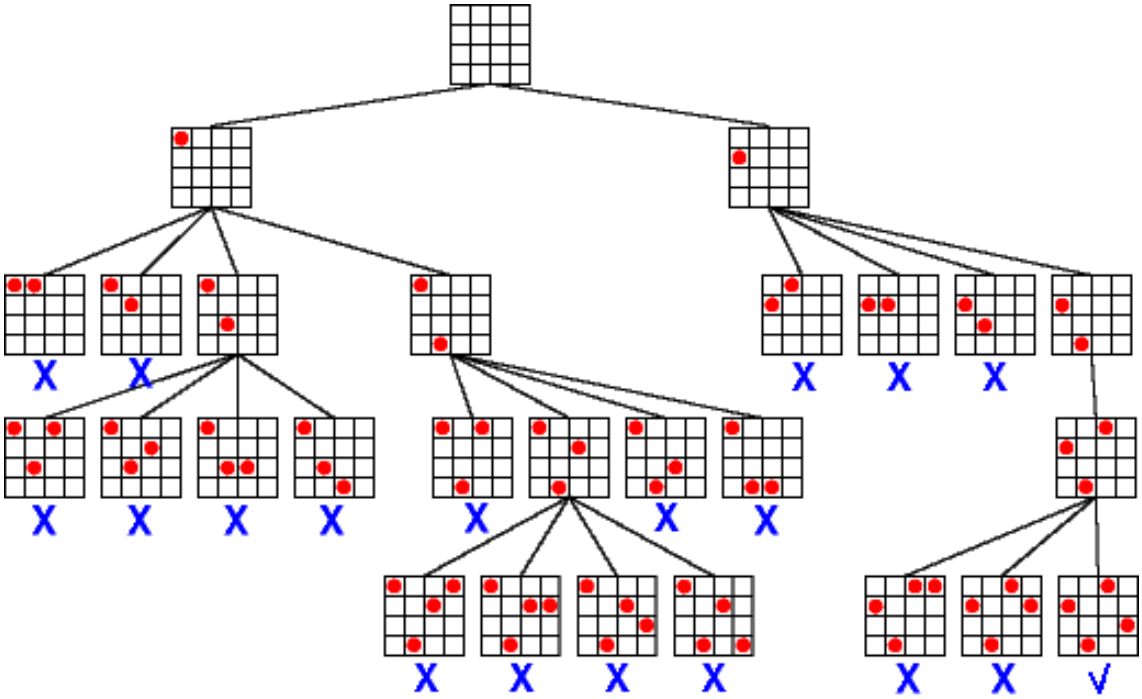

Abbildung 3.2: Suchbaum für das 4-Damen-Problem[9]

Suche weniger Permutationen betrachtet werden müssen. Constraint Propagation kann und wird nach jedem Schritt des Backtracking-Algorithmus angewendet, um die durch die Zuweisung eines "konstanten" Wertes neu gewonnenen Informationen im CSP widerzuspiegeln.

Ausgehend von diesen Ansätzen wurden zahlreiche weiterführende Algorithmen entwickelt, wie z.B. *backjumping* und *forward-checking* [15].

Constraint Satisfaction Probleme finden Anwendung in der realen Welt. Hierzu gehören vor allem verschiedene NP-vollständige Optimisierungsprobleme.

3.3.2 Im Bezug zu Szenarien

Das Problem der Szenarioverwaltung hat einige Gemeinsamkeiten mit einem anderen "real world" CSP, dem weit bekannten Course Scheduling Problem[11]. Beim Course Scheduling geht es darum eine Reihe von Kursen auf eine Reihe von Lehrern in bestimmte Zeiträume zu verteilen. Dabei können bestimmte Lehrer nur bestimmte Kurse leiten, jeweils nur einen Kurs auf einmal und nicht mehr als eine bestimmte Anzahl pro Woche.

Die Szenarioverwaltung in Qivicon wird zum Teil mit vergleichbaren Problemen konfrontiert. Es gibt Szenarien, die eine Zuordnung von einer Reihe von Gerätezuständen zu bestimmten Zeiträumen darstellen. Dabei dürfen Szenarien sich nicht überschneiden (sie dürfen in so einem Fall nicht die gleiche Priorität haben). Da Szenarien durch den Be-

nutzer manuell definiert werden, müssen keine Zeitbereiche automatisch berechnet werden. Die Geräte und deren entsprechenden Zustände werden offensichtlich auch von dem Nutzer direkt angegeben. Als Constraint Satisfaction Problem kann dabei vor allem aber die Prioritätsvergabe angesehen werden.

Bei der Betrachtung des formalen Modells eines Szenarios aus Anfang von Kapitel 3 wird deutlich, dass T und G konstant sind und nur für P automatisch Werte zugewiesen werden sollen. Dabei kann man zwischen *impliziten* und *expliziten* Constraints unterscheiden. Unter "explizite Constraints" sind die durch den Nutzer manuell über die GUI definierten Einschränkungen gemeint. Die impliziten hingegen sind die default-Relationen zwischen den Szenen (z.B. eine später definierte Szene hat eine höhere Priorität als ältere Szenen).

Es ist gewollt, dass bei der Erstellung und Löschung einer Szene das System auf Konflikte geprüft wird und die existierenden Prioritäten neu kalkuliert werden. Daher muss bei jeder Änderung der Menge der Szenarien das CSP erneut, mit mehr (bzw. weniger) Variablen und Constraints gelöst werden.

3.3.3 Fazit

Vor allem die Prioritätenvergabe, die den Einsatz der anderen Ansätze unangemessen machte, lässt sich gut als Constraint Satisfaction Problem lösen. Eine Anwendung von CSPs in diesem Kontext ist daher angebracht. Die Konfliktfindung muss jedoch mit einem eigenen, dedizierten Algorithmus (ähnlich zu Sektion 3.1.2) realisiert werden.

Kapitel 4

Entwurf

Bevor mit der Implementierung angefangen werden kann, müssen erst Entscheidungen getroffen werden, *wie* die einzelnen Anforderungen umgesetzt werden sollen. Es folgt eine schrittweise Analyse der einzelnen Anforderungen.

4.1 Definitionen

Als Erstes werden einige Definitionen eingeführt. Es muss festgelegt werden, was genau ein Szenario ist und wann ein Konflikt vorliegt.

4.1.1 Szenario

Ein Szenario ist eine Umsetzung des in Kapitel 3 vorgestellten Tupels. Es besteht aus dem, *was* getan werden muss und aus dem, *wann* es getan werden muss.

Um eine bestimmte Aktivitätsdauer festzulegen, müssen Start- und Endzeitpunkte gespeichert werden. Außerdem wird ein bestimmtes Wiederholungsintervall benötigt.

Zielzustände werden benutzt, um für die betroffenen Geräte eindeutig festlegen, welches Verhalten sie aufweisen sollen.

Schließlich gibt es noch die Priorität, die Konfliktfälle löst.

4.1.2 Zustand

Es gibt zwei Arten von Zielzuständen. Einerseits die gewöhnlichen, die einer Eigenschaft eines Geräts einen bestimmten Wert zuweisen. Andererseits existieren die Minimum-/Maximum-Zustände, die für eine Eigenschaft eines Gerätes Grenzen definieren, sodass der aktuelle Zustand des Gerätes nicht den entsprechenden Minimum-Wert unterschreiten, bzw. den Maximum-Wert überschreiten darf.

4.1.3 Konflikt

Ein Konflikt zwischen zwei Szenarien liegt vor, wenn sie für den **gleichen** Zeitpunkt **verschiedenes** Verhalten für **ein** Gerät definieren.

4.2 Ziel

Allgemein betrachtet muss das resultierende Bundle Methoden zur Szeneverwaltung bereitstellen. Konkreter gesehen, müssen 6 Funktionalitäten geboten werden: Speichern, Löschen, Starten, Stoppen, Aktivieren und Deaktivieren. Die ersten vier Begriffe sind selbsterklärend. Ein Szenario wird als "deaktiviert" bezeichnet, wenn es zwar existiert, aber nicht ausgeführt werden soll.

4.3 Ideen

4.3.1 Szenario

Als erstes muss entschieden werden, wie genau ein zeit-basiertes Szenario mit den vorhandenen Mitteln realisiert werden kann.

Die Start- und Endzeitpunkte lassen sich leicht durch die existierenden Conditions realisieren. Jeweils eine *AbsoluteTimerCondition* zu den beiden Zeitpunkten ermöglicht es einerseits, am Anfang die Ausführung des Szenarios zu gewährleisten, andererseits das Ende der Aktivitätsdauer zu signalisieren. Alternativ wäre diese Logik auch mit Java-Timern umsetzbar.

Es ist zu entscheiden, ob mit absoluten Zeitangaben (an ein konkretes Datum gebunden) gearbeitet wird oder doch relative (z.B. jeden Dienstag, etc.) eine höheres Maß an Flexibilität bieten.

Die tatsächliche Schaltung der Zustände kann prinzipiell auf zwei Arten realisiert werden. Einerseits ist es möglich mit Hilfe von Commands die jeweiligen Geräte zu kontrollieren. Es wird eine Rule erstellt, die zur Startzeit die entsprechenden Geräte steuert. Andererseits können die Geräte auch direkt programmatisch gesteuert werden. Eine entsprechende API wurde mit der Qivicon Homebase mitgeliefert.

Leider lassen sich Szenarien nicht komplett durch existierende Elemente realisieren, was dazu führt, dass in jedem Fall viele Informationen im Domänenmodell gespeichert und verwaltet werden müssen. Dafür gibt es eine Reihe von Gründen:

1. Wenn ein Szenario endet, müssen andere Szenarien ausgeführt werden. Daher werden Data-Container-Objekte (DCO) benötigt, die die gesamten Informationen über ein Szenario speichern. Sonst ist es nicht möglich festzustellen, welche Szenarien gerade aktiv sind und entsprechend ausgeführt werden müssen.

2. Die Zeit, während der das automatische Schalten von Geräten verhindert wird. Um eine solche Sperre zu realisieren, braucht man für jedes Gerät ein es repräsentierendes DCO, wo sie gespeichert werden kann.

Die Tatsache, dass innerhalb des Szenemanagement-Bundles alle relevanten Informationen in Form von DCOs verwaltet werden müssen, führt zu der Frage, ob es nicht sinnvoll ist, sich komplett von den existierenden HAM-Tools zu distanzieren und die gesamte Verwaltung durch interne Tools zu regeln.

Umsetzungskonzept

Im Rahmen dieser Arbeit wurde die Entscheidung getroffen, dass die Zeitsteuerung mit Hilfe von HAM realisiert wird. Es werden Timer-Conditions genutzt, um zum Start- und Endzeitpunkt jeweils ein Event zu publizieren. Außerdem sollen Custom Commands erstellt werden, die ein Szenario jeweils starten und stoppen können. Mit Hilfe von Rules sollen die Befehle an die entsprechenden Conditions gebunden werden.

Die Zeitangabe wird in absoluten Werten abgehandelt, um die Berechnung der Konflikte zu vereinfachen; vor allem bei mehrtägigen Szenen würde der entsprechende Code andererseits sehr kompliziert werden.

4.3.2 Prioritäten

Prioritäten werden benötigt, um Konflikte zwischen Szenarien zu lösen. In einem Konfliktfall wird die Szene mit der höheren Priorität ausgeführt. Da, wie in Sektion 4.3.1 erläutert, jedes Szenario ein zugehöriges DCO haben wird, kann die Priorität einfach als zusätzliches Integer-Attribut gespeichert werden.

Die Vergabe der Prioritäten wird als folgendes Constraint Satisfaction Problem dargestellt.

Variable

Wie in Sektion 3.3.2 erläutert, werden nur die Prioritäten als Variablen dargestellt. Daher enthält das Problem eine variable, aber endliche Menge von Integer-Variablen, die den Prioritäten entsprechen.

Domain

Es ist festgelegt, dass es nur eine begrenzte Anzahl, also n, verschiedene Prioritäten geben soll. Daher hat die Variable *Priorität* eine Domain von 1 bis n. Eine solche Begrenzung führt dazu, dass für einen beliebigen Zeitpunkt t nie mehr als n verschiedene Szenarien einen bestimmten Zielzustand für *ein* Gerät definieren.

Constraints

Das grundlegende Constraint ist, dass zwei sich zeitlich überlappende Szenarien, falls sie in Konflikt stehen, nicht die gleiche Priorität besitzen dürfen. Die default-Eingrenzung ist, dass, falls vom Benutzer nicht anders definiert, ein später erstelltes Szenario eine höhere Priorität hat, als ältere.

Im Rahmen dieser Arbeit wird die Constraint Satisfaction nur prototypisch für derartige simple Constraints implementiert. Eine Erweiterung um komplexe Constraints könnte in weiterführenden Arbeiten in der Zukunft behandelt werden.

4.3.3 Kein Endzustand

Eine Anforderung deren Erfüllung trivial ist, denn es muss nichts gemacht werden. Sobald der Endzeitpunkt erreicht wird, gilt das Szenario als inaktiv, ohne das etwas anderes gemacht wird.

4.3.4 Erneute Ausführung

Eine Anforderung, die nur auf Code-Ebene realisierbar ist. Wenn ein Szenario endet, werden die existierenden Szenarien analysiert und die mit der höchsten Priorität werden nochmal ausgeführt. Da man in jedem Fall DCO-Objekte für die Szenarien hat, wie in 4.3.1 erklärt, müssen an dieser Stelle keine neuen Komponenten eingeführt werden.

Die erneute Ausführung kann logisch auf zwei Arten realisiert werden:

1. Man kann einfach alle aktiven Szenarien in Reihenfolge von niedrig bis hoch nacheinander erneut starten. Dadurch würden alle Geräte den für den aktuellen Zeitpunkt definierten Zustand erreichen, aber so ein Vorgehen würde zu wiederholtem Zwischenschalten führen, sodass bei einer großen Anzahl von aktiven Szenarien der Nutzer ein ungültiges Verhalten des Systems bemerken könnte.

2. Alternativ kann auch eine Analyse der aktiven Szenarien durchgeführt werden, mit dem Ziel nur die relevanten Zielzustände für alle Geräte zu identifizieren. Danach müsste man nur die Geräte auf diese Werte schalten.

Umsetzungskonzept

Da es bei der 2. Variante zu keinem zwischenzeitlichen Fehlverhalten des Systems kommt, wird im Rahmen dieser Arbeit diese Idee implementiert.

4.3.5 Manuelle Steuerung

Das manuelle Steuern der Geräte ist selbstverständlich jederzeit möglich. Das Übersteuern der gewöhnlichen Zustände stellt in diesem Fall kein Problem dar, weil keine Systemreak-

tion erforderlich ist. Im Falle der Minimum-/Maximum-Zustände muss jedoch nach dem manuellen Schalten der aktuelle Wert ausgelesen und, falls er außerhalb der Grenzen liegt, korrigiert werden.

Hier ist von Nutzen, dass die Qivicon Homebase bei jeder Änderung eines Zustands eines Gerätes ein "DCO"-Event wirft, in dem das betroffene Gerät und der neue Zustand eindeutig identifiziert werden. Mit Hilfe eines entsprechenden Event Listeners lassen sich diese Events auffangen und, falls nötig, kann entsprechend reagiert werden.

4.3.6 Sperren

Um das Blockieren von Geräten zu realisieren, muss man zunächst eine Möglichkeit finden zu unterscheiden, ob ein Gerät manuell geschaltet wurde oder durch ein Szenario. Leider unterscheiden sich die DCO-Events, die in 4.3.5 vorgestellt wurden, nicht in Abhängigkeit davon, ob sie durch einen Menschen direkt oder durch ein Szenario ausgelöst wurden. Sie reichen daher nicht aus um eine derartige Differenzierung durchzuführen.

Dennoch lassen sich diese beiden Fälle mit nahezu 100% Wahrscheinlichkeit unterscheiden. Da man zur Laufzeit Informationen über alle existierenden Szenarien besitzt, kann man vorhersagen, wann ein durch ein Szenario ausgelöstes Event erwartet wird.

Um festzustellen, welcher Fall vorliegt, muss man daher nach dem Auffangen des Events prüfen, ob der neue Wert mit dem *erwarteten* Wert übereinstimmt. Hierzu muss vor jedem Schalten durch ein Szenario für jedes Gerät gespeichert werden, welcher Wert erwartet wird. Dies lässt sich leicht in den DCOs (siehe Sektion 4.3.1) für die jeweiligen Geräte realisieren.

Nachdem eine derartige Unterscheidung realisiert wurde, muss auch hier entschieden werden, ob man die Sperrzeit in absoluten oder relativen Werten angibt. Da aber in Sektion 4.3.1 schon die absoluten Zeitangaben gewählt wurden, kommen sie auch hier zum Einsatz.

4.3.7 Minimum-/Maximum-Zustände

Das Definieren von Grenzwerten kann ebenfalls nur auf der Code-Ebene stattfinden, da Qivicon keine Tools hierfür bietet. Es existiert jedoch eine API, die es erlaubt die aktuellen Zustände von Geräten abzufragen. In Verbindung mit den DCO-Events lassen sich die aktuellen Zustände der Geräte jederzeit verfolgen.

Da es leider keine Möglichkeit gibt das manuelle Setzen eines Attributs auf einen Wert außerhalb des Grenzwerts zu verhindern, muss eine schnellstmögliche Korrektur des Wertes ausreichen. In dem Moment, wo ein Minimum-/Maximum-Zustand aktiviert wird, wird der aktuelle Zustand des Geräts überprüft und, falls nötig, korrigiert. Bei beliebigen Änderungen des Zustands in der Zukunft werden die geworfenen DCO-Events analysiert und die Zustände gegebenenfalls auf die Grenzwerte zurückgesetzt.

Es ist außerdem zu entscheiden, ob auf der Szenarienebene zwischen den Zielzuständen unterschieden wird (z.B. *GeneralScene* und *MinMaxScene*) oder ob es besser ist, nur einen Typ von Szenen zu haben und die Unterscheidung auf Zustandsebene durchzuführen.

4.4 Fazit

Die im Rahmen des Smart Home gesteuerten Geräte werden mit Hilfe von Data Container Objekten innerhalb des Bundles widergespiegelt. Diese DCOs werden mit den realen Geräten, bzw. den ihnen entsprechenden Qivicon-DCOs, synchronisiert. Die Steuerung erfolgt durch Änderung der entsprechenden Attribute der DCO-Objekte.

Kapitel 5

Implementierung

Die Implementierung wurde entlang des Entwurfs durchgeführt. Es werden UML- und Sequenz-Diagramme verwendet, um alles zu visualisieren. Da die Qivicon-Box zurzeit leider nur Java 1.4 unterstützt, musste auf die aktuelleren Funktionalitäten der Sprache verzichtet werden.

5.1 Struktur

Es wurde versucht möglichst generisch zu implementieren. Allgemein betrachtet lässt sich die Implementierung in Modell und Controller aufteilen. Eine View ist nicht Teil dieser Arbeit.

5.1.1 Model

Scene

Wie auf der Grafik 5.1 zu sehen ist, gibt es zwei wichtige Modell-Klassen. Eine davon ist die *Scene*, die die Szenarien darstellen soll. Sie enthält die im Kapitel 4 diskutierten Attribute, sowie zuzüglich eine Reihe von Verwaltungsinformationen (z.B. eine eindeutige Id), die im Rahmen dieser Arbeit nicht von Interesse sind.

Abstract Target State

Eine Szene enthält eine beliebige Anzahl von Zielzuständen (Target States). Es wird unterschieden zwischen den generellen Zielzuständen und den Minimum-Maximum-Zuständen. Eine derartige Struktur wurde gewählt, weil sie es erlaubt Szenarien zu definieren, die sowohl feste Zielwerte, als auch Grenzwerte enthalten, ohne eine Differenzierung auf Szenarien-Ebene vorzunehmen. Das macht den Aufbau generischer und erlaubt es in Zukunft weitere Typen von Zielzuständen zum System hinzuzufügen, ohne viel ändern zu müssen.

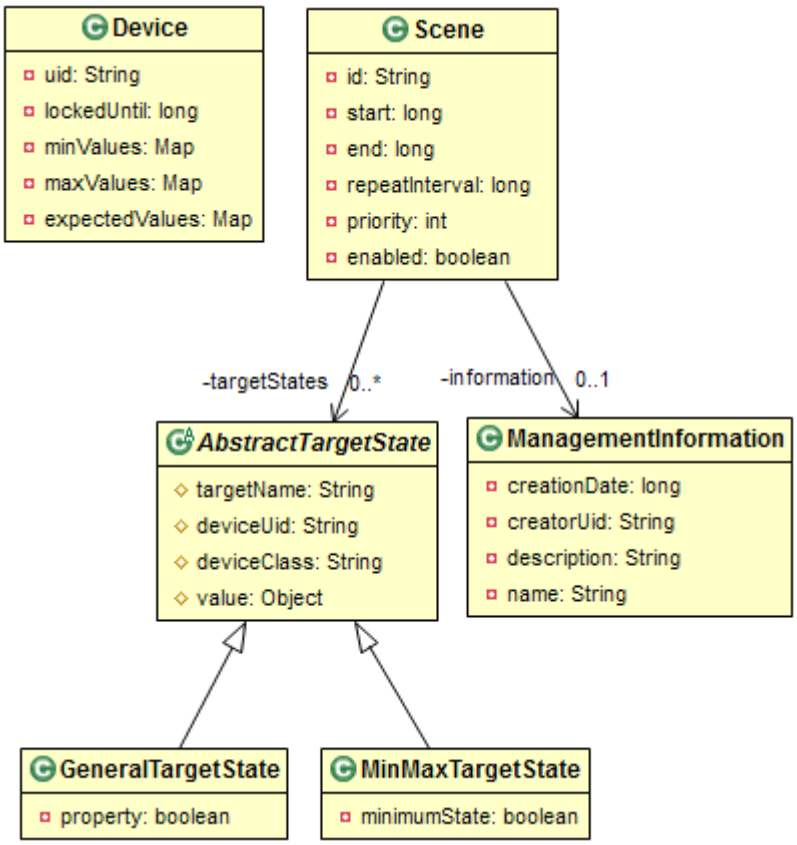

Abbildung 5.1: Die Model Klassen

Das Ausführen eines Szenarios besteht im Grunde nun aus dem schrittweisen Ausführen jedes einzelnen Zielzustands. Ein derartiger modularer Aufbau ermöglicht es außerdem, dass vor dem Ausführen eines Zielzustands eine Analyse stattfinden kann, ob dieser Zustand tatsächlich relevant ist oder ob ein aktives Szenario mit einer höheren Priorität bereits einen anderen Zielzustand für das entsprechende Gerät definiert. Dadurch lassen sich redundante Schaltungen, wie in Sektion 4.3.4 dargestellt, vermeiden. Ein Szenario schaltet tatsächlich nur die Geräte, für die es in diesem Moment die höchste Priorität besitzt.

Als logische Folge eines solchen Aufbaus kann eine Szene eine beliebige Mischung aus verschiedenen Zustandstypen enthalten. Es ist lediglich zu beachten, dass *ein* Szenario keine sich widersprechende Zielzustände haben darf, was mit Hilfe des Validierens verhindert wird.

Device

Die Modell-Klasse Device ist eine Repräsentation der existierenden Geräte. Jedes Objekt dieser Klasse spiegelt ein durch Qivicon gesteuertes Gerät wieder, dass an Hand seiner Uid eindeutig identifizierbar ist.

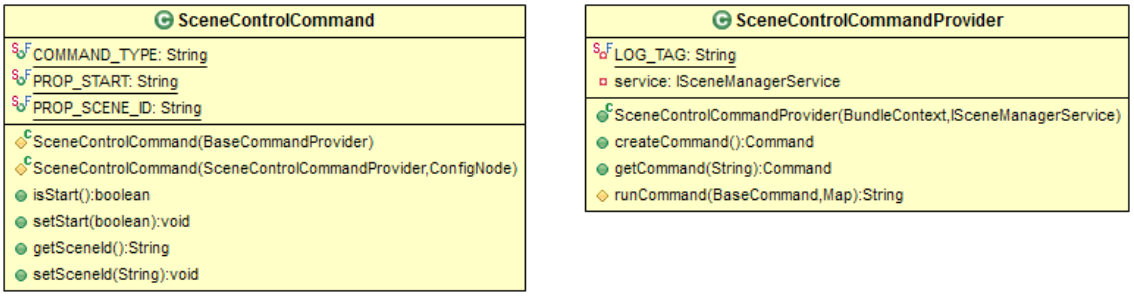

Abbildung 5.2: Die custom Command zur Steuerung von Scenen

In diesen Objekten werden einerseits die durch Minimum-Maximum-Zustände definierten Grenzwerte gespeichert, andererseits wird hier das Sperren der Geräte realisiert. Außerdem werden an dieser Stelle die erwarteten Werte gespeichert, die das Differenzieren zwischen manueller und automatischer Schaltung ermöglichen.

Nach jedem manuellen Schalten wird ein absolutes Datum innerhalb der entsprechenden Instanz gespeichert, mit dem jedes Szenario vor dem tatsächlichen Schalten des Gerätes einen Abgleich durchführt. Falls dieses Datum noch in der Zukunft liegt, gilt das Gerät als gesperrt und wird nicht geschaltet.

Scene Control Command

Bei der *SceneControlCommand* handelt es sich um eine Implementierung der in Sektion 4.3.1 diskutierten Elemente (Abbildung 5.2). Es wurde das von Qivicon bereitgestellte Interface zur Erstellung von Custom Commands benutzt.

Jede *SceneControlCommand* identifiziert per Id die betroffene Szene und unterscheidet, ob es sich um einen Start- oder Endbefehl handelt. Der *CommandProvider* ruft bei Aktivierung des entsprechenden Befehls über das Interface (siehe Sektion 5.1.2) die zugehörige Aktion auf.

5.1.2 Controller

Die Controller sieht man am besten an der Grafik 5.3. Es folgen die einzelnen Komponenten im Detail:

ISceneManagerService

Das zentrale Interface für das gesamte Programm ist der ISceneManagerService. Es enthält die Methoden, deren Implementierung als Service für die anderen OSGi-Bundles bereitgestellt wird. Ihrerseits benutzt die Implementierung verschiedene existierende Services anderer Bundles.

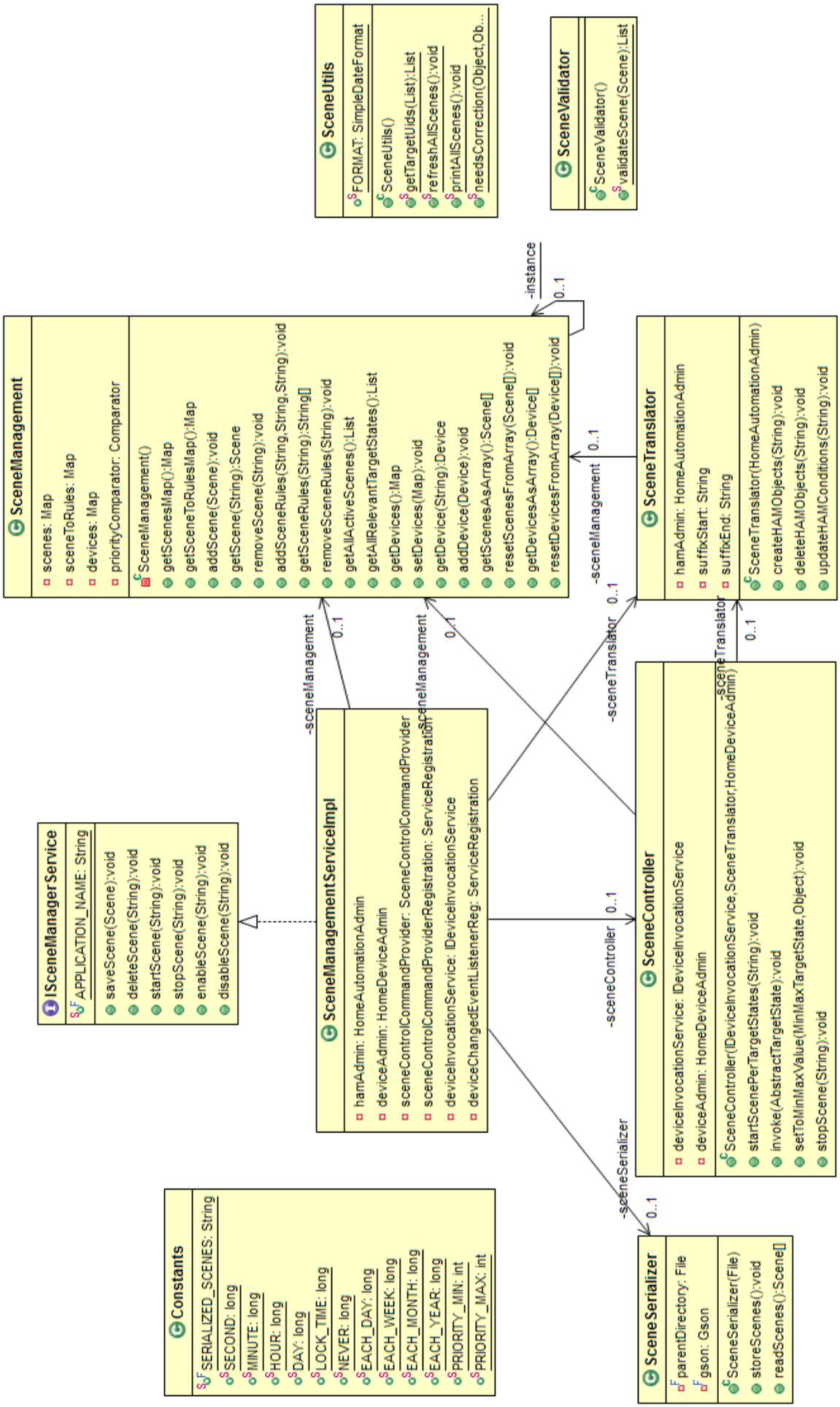

Abbildung 5.3: Ein Überblick über die Implementierung

Scene Translator

Im SceneTranslator werden die zugehörigen HAM-Objekte zu einer Szene erstellt. Bei dem Speichern einer Scene werden zwei aboslute Timer gesetzt. An diese Conditions wird per Rule jeweils ein *SceneControlCommand* gebunden.

Scene Management

Das *SceneManagement* ist der zentrale Datenverwaltungspunkt des Bundles. Es wurde als Singleton realisiert und beinhaltet alle Scenen, die zu ihnen gehörenden Start- und Stopp-Rules und die Devices.

Außerdem bietet die Klasse verschiedene Methoden zum Abruf von Untermengen der gesamten Daten. Dank der Umsetzung des Singleton-Entwurfsmusters sind die Informationen immer aktuell und alle anderen Klassen können jederzeit bequem auf die von ihnen benötigten Informationen zugreifen.

Scene Controller

In dem *SceneController* wurde die tatsächliche Logik des Starten und Stoppen von Szenarios realisiert.

Wie im Entwurf geplant, wurde die tatsächliche Steuerung durch Ausführen von Target States umgesetzt, was die Methode *invoke(AbstractTargetState)* (siehe Abbildung 5.3) reflektiert. Sowohl beim Starten, als auch beim Stoppen einer Szene findet eine Analyse aller aktiven Szenen statt. Es werden nur die relevanten Zielzustände gesammelt und ausgeführt.

Scene Serializer

Das Persistieren der Szenen findet in Form eines serialisierten JSON Strings statt. Mit Hilfe von Google GSon[18], einer Bibliothek, die unter anderen ebenfalls von Qivicon bereitgestellt wird, erfolgt die Serialisierung.

Da leider mit Java 1.4 gearbeitet werden musste, wird für eine erfolgreiche Serialisierung der *SceneSerializer* benötigt. Da die Typisierung von Collections erst in Java 1.5 eingeführt wurde, muss mit Hilfe dieses Serializers und den zusätzlichen Meta-Informationen, die er in die entstehende Datei schreibt, unterschieden werden, um welche Art von Objekt es sich jeweils handelt. Ansonsten könnte GSon z.B. beim Deserialisieren von *Abstract Target States* nicht auseinander halten, ob es sich unter dem konkreten Objekt um einen *MinMax*- oder einen *GeneralTargetState* handelt.

Scene Validator

Bevor eine Szene tatsächlich gespeichert wird, muss noch eine Validierung der Attribute stattfinden, um sicherzustellen, dass keine Probleme aufgrund ungültiger Eingaben entstehen. Der Validator gibt eine Liste von *Validation Errors* zurück. Falls diese Liste leer ist, wird die Szene gespeichert. Ansonsten wird die Liste der Errors weitergereicht, bis sie z.B dem Nutzer im entsprechenden User Interface vorgestellt werden.

Der Validator ist dank der Existenz von Prioritäten relativ trivial ausgefallen. Es wird geprüft, ob alle Attribute einer Scene nicht **null** sind. Außerdem wird kontrolliert, ob die Dauer einer Szene tatsächlich kleiner ist, als die Wiederholungsperiode. Schließlich wird analysiert, ob ein Szenario keine in sich widersprüchlichen Target States besitzt.

Scene Utils

In den SceneUtils ist eine Reihe von verschiedenen Hilfsmethoden gelagert, die von mehreren Klassen benötigt werden.

Constants

In der Klasse Constants sind, wie der Name schon sagt, verschiedene globale Konstanten gelagert. Hier wird die Sperrzeit t eingestellt, während der Geräte nach einer manuellen Schaltung nicht durch Szenarien geregelt werden können. Außerdem befinden sich hier die Codes für die verschiedenen Wiederholungsintervalle, der Definitionsbereich der Prioritäten und verschiedene Zeitintervalle in Millisekunden.

5.1.3 Constraint Satisfaction

Wie geplant, wurde bei der Implementierung ein existierendes Framework zum Erstellen und Lösen von Constraint Satisfaction Problems verwendet. Es wurde die Bibliothek "Choco Solver 3" verwendet.

Choco Solver

Das Framework Choco Solver[3][12] ist ein Open Source Java Bibliothek zur Lösung von Constraint Satisfaction Problemen. Es bietet eine Reihe von vorgefertigten Variablen, Constraint-Typen und Lösungsstrategien, mit denen relativ simple CSPs problemlos erstellt und gelöst werden können.

Die Funktionsweise des Solvers ist wie folgt:

1. Erstelle einen Solver.
2. Erstelle die Variablen mit Hilfe der entsprechenden Factory.
3. Erstelle die Constraints mit Hilfe der entsprechenden Factory.

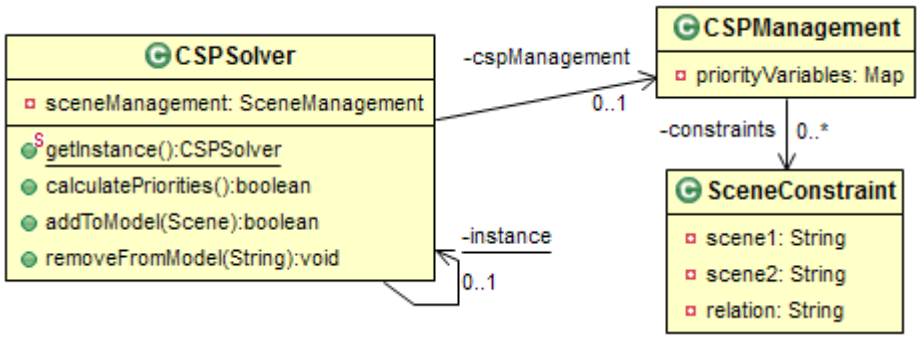

Abbildung 5.4: Die Einbindung von Choco Solver

4. Wähle/Definiere die Lösungsstrategie.

5. Starte den Solver.

Es ist zu beachten, dass sobald der letzte Schritt ausgeführt wurde, das Problem als geschlossen gilt. Es ist nicht möglich, nachdem eine Lösung gefunden wurde, weitere Variablen und Constraints zum existierenden Problem hinzuzufügen und es noch einmal lösen zu lassen. Um also die Daten wiederzuverwenden, muss man eine Kopie des Problems erstellen, mit neuen Objekten.

Realisierung

Die Art und Weise, auf die der Choco Solver in das Szenenmanagement eingebunden wurde, lässt sich gut anhand von Abbildung 5.4 darstellen.

Die Relationen zwischen den Szenen, die in Form von Constraints realisiert sind, werden in Form von *SceneConstraints* gespeichert. Für die Verwaltung dieser Objekte ist die Klasse *CSPManagement* verantwortlich. Die Schnittstelle, über die auf die CSP-Elemente der Implementierung von den anderen Komponenten aus zugegriffen wird, ist der *CSPSolver*. Er ist als Singleton realisiert und bietet die Möglichkeit, jeweils eine Szene zum CSP hinzuzufügen oder zu entfernen. Es gibt auch die Möglichkeit, die existierenden Prioritäten neu zu kalkulieren.

Das Constraint Satisfaction Problem wird induktiv aufgebaut. Wenn es keine Szenen gibt, dann gibt es auch keine Constraints. Mit jeder hinzukommender Szene findet eine Prüfung statt, ob sie in Konflikt mit einer der bisher existierenden Szenen steht. Falls nein, so werden keine neuen Constraints generiert, andernfalls wird, abhängig von der Erstellungsdatum der Szenen das Constraint erstellt, dass die Priorität der jüngeren größer ist, als die der älteren. Sobald die Constraints alle erstellt wurden, wird das CSP anhand des in Sektion 5.1.3 beschriebenen Vorgehens generiert und gelöst. Gibt es eine Lösung, so werden alle Szenen mit den neuen Prioritäten aktualisiert. Falls nicht, so findet ein Rollback statt, die gerade gespeicherten *SceneConstraints* werden gelöscht und es wird eine Fehlermeldung zurückgegeben.

Beispiel

Um die Zusammenarbeit der einzelnen Komponenten zu visualisieren, wurde die Vorgehensweise beim Speichern einer Szene schematisch in einem Sequenzdiagramm (Abbildung 5.6) dargestellt.

Wenn ein Nutzer eine Szene erstellt, wird sie in dem *SceneManagementServiceImpl* verarbeitet. Dort wird sie zunächst validiert. Falls es keine ValidationErrors gibt, wird die Szene zum CSP hinzugefügt. Nachdem sie vom *CSPSolver* eine Priorität erhalten hat, wird mit Hilfe des Wiederholungsintervalls die nächste Ausführungszeit bestimmt und die HAM-Objekte werden angelegt. Abschließend wird die erstellte Szene in das *SceneManagement* aufgenommen.

Falls die Validierung fehlschlägt, entfallen die folgenden Schritte und die ValidationErrors werden einfach weitergereicht (siehe Abb. 5.5).

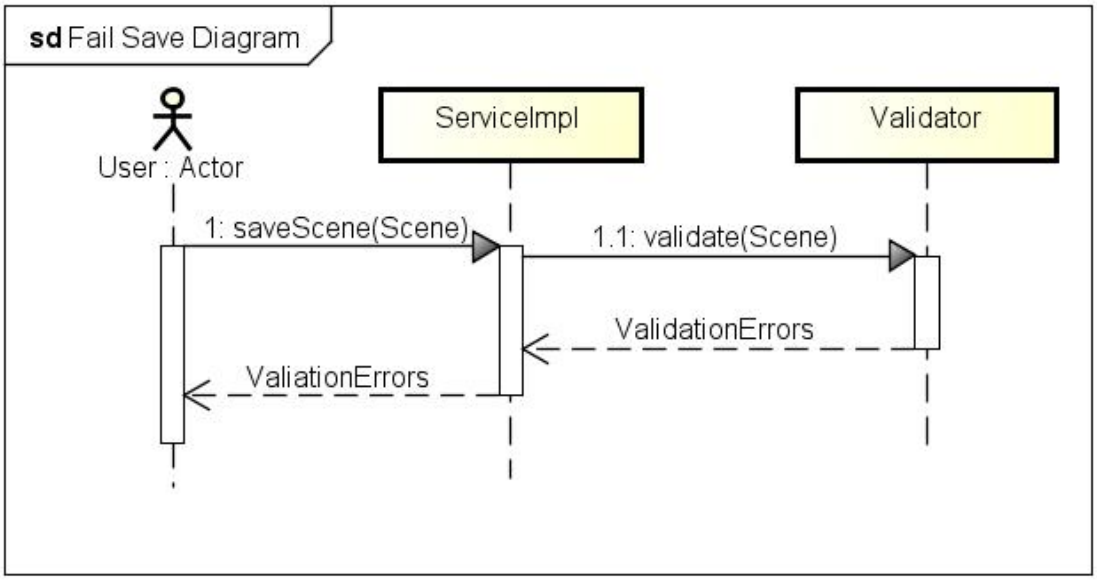

Abbildung 5.5: Fehlgeschlagene Validierung beim Speichern einer Szene

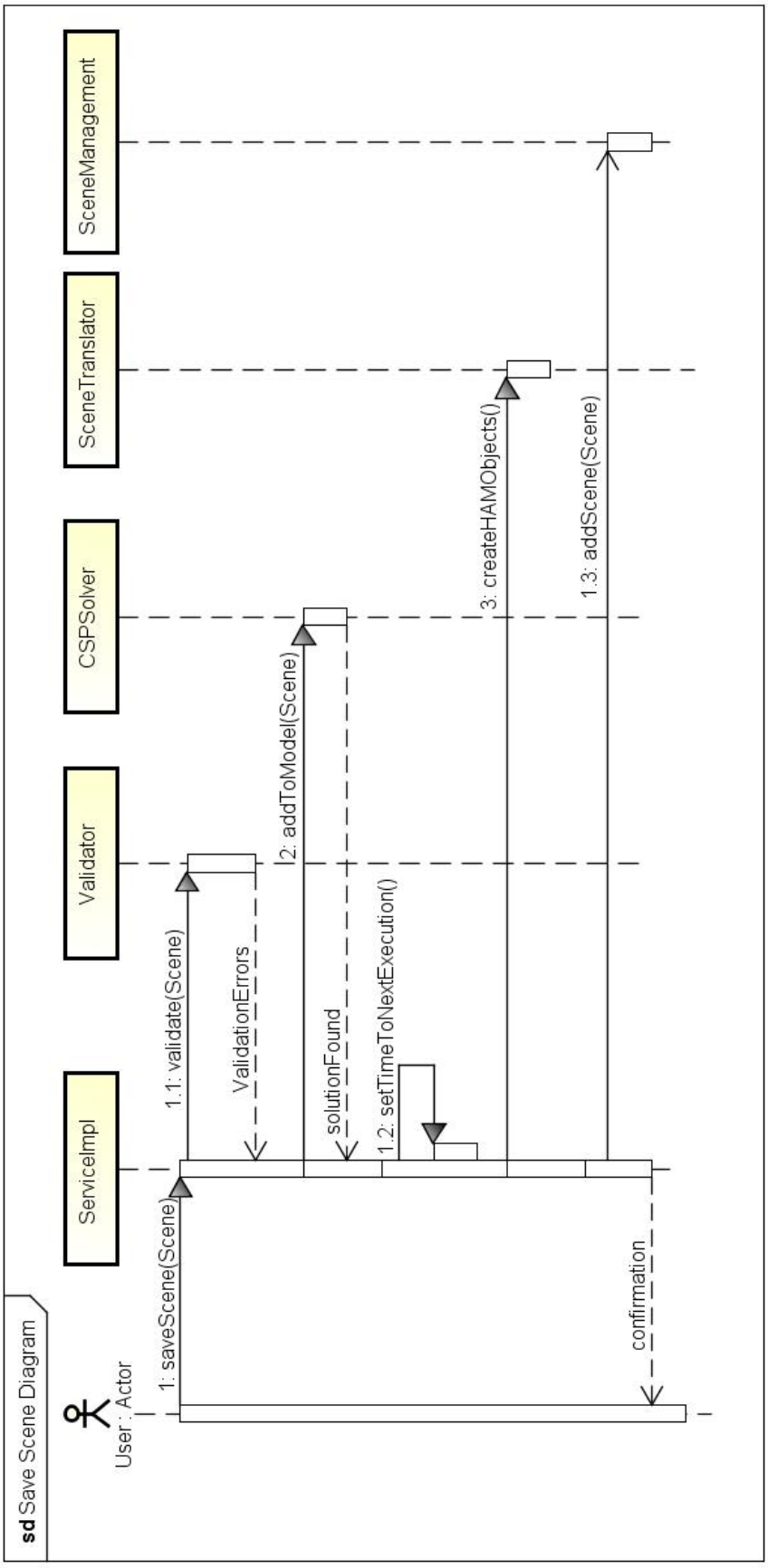

Abbildung 5.6: Erstellen einer Szene als Sequenzdiagramm

Kapitel 6

Evaluation

Nach der Implementierung muss noch der wahrscheinlich wichtigste Punkt abgehandelt werden: Die Evaluation.

6.1 Erfüllte Ziele

Alle formalen Anforderungen, wie sie in Sektion 2.2.4 erläutert sind, wurden erfüllt. Es lassen sich zeit-basierte Szenen mit unterschiedlichen Prioritäten definieren, sodass Geräte automatisch zu einer bestimmten Zeit ihre Zustände verändern. Bei Ende eines aktiven Szenarios werden für jedes Gerät die Szenen mit den nächst-höchsten Prioritäten neu aktiviert. Manuelle Steuerung von Geräten hat höhere Priorität, als die des allgemeinen Schaltens und sperrt das betroffene Gerät für eine bestimmte Zeit für autonome Änderungen.

Die Implementierung wurde einem umfassenden Test unterzogen. Es wurden verschiedene Szenen mit puren Minimum-Maximum-Zielzuständen erstellt, gewöhnliche Szenen mit konkreten Wertzuweisungen und gemischte Szenen, wo beide Typen von Zielzuständen beliebig gemischt wurden. Das erwartete Verhalten des Systems wurde mit dem tatsächlichen verglichen und auf Diskrepanzen überprüft.

Die meisten Tests wurden auf einem Emulator auf einem Rechner mit 2 Prozessoren mit je 2 Ghz und 4 GB RAM ausgeführt. Allerdings funktioniert die Implementierung auch auf den Produktivboxen, die bei Anasoft Technology AG stehen.

Es stellte sich heraus, dass die Reaktion des Systems in der absoluten Mehrheit der Fälle korrekt war. Das selten auftretende Fehlverhalten lässt sich auf das in Sektion 6.3.2 erläuterte Problem zurückführen, sowie auf hin und wieder im Emulator auftretende interne Fehler. Die letzteren Fälle waren nie replizierbar und ließen sich stets durch einen Neustart des Emulators beheben.

Zusätzlich zu den allgemeinen Funktionalitäten wurde die Leistungsfähigkeit der CSP-Elemente in der nachfolgenden Sektion geprüft.

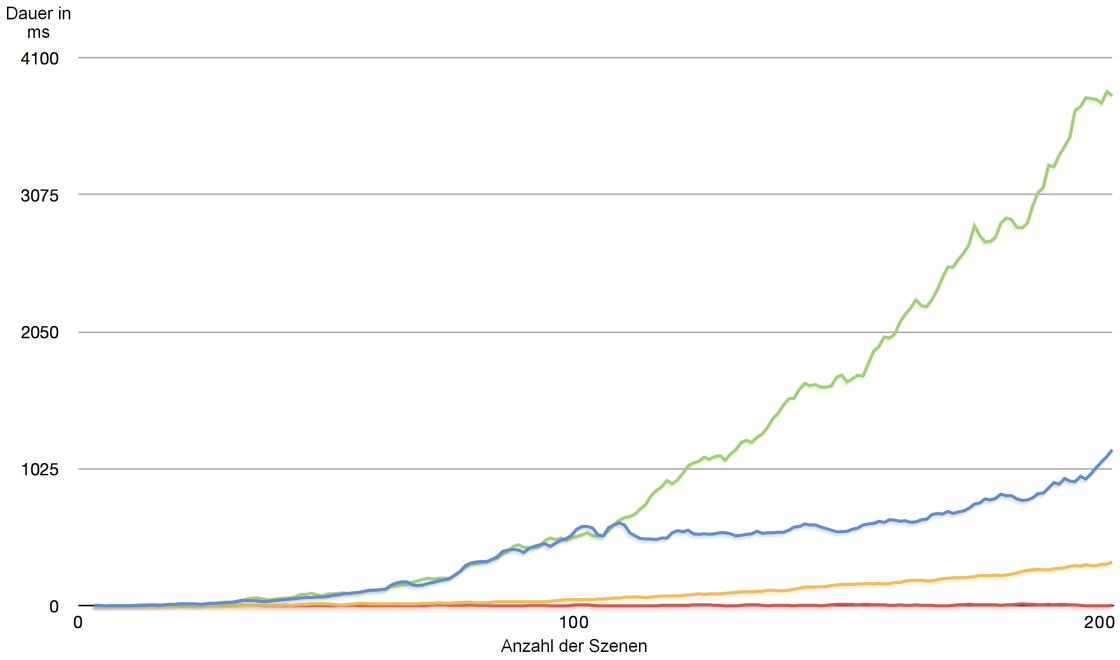

Abbildung 6.1: Dauer der Neuberechnung der Prioritäten pro Szene in Millisekunden. Grün dargestellt ist der ungünstigste Fall mit maximaler Anzahl von Konflikten. Rot ist die minimale Anzahl von Konflikten. Blau sind 2 Konflikt-Blöcke mit je 100, gelb 4 Blöcke mit je 50 Szenen dargestellt.

6.2 Constraint Satisfaction Problem

Im Abbildung 6.1 ist die Dauer der kompletten Neuberechnung aller Prioritäten bei Erstellung einer neuen Szene dargestellt. Darin inbegriffen ist das Erstellen und Lösen des CSP, sowie die Aktualisierung aller Prioritäten in den Scene-Objekten. Auf der x-Achse ist die Anzahl der Szenen und auf der y-Achse die Dauer in Millisekunden angegeben.

Es wurden jeweils 200 Szenen erstellt und getestet. Im ungünstigsten Fall (grüner Graph) waren alle Szenen in Konflikt miteinander. Wie leicht zu sehen ist, steigt die Dauer in so einem Fall exponentiell an; glücklicherweise liegt die erwartete Menge von sich überschneidenden Szenen deutlich unter hundert, wo die Dauer unter einer Sekunde beträgt.

Der rote Graph stellt den gegenteiligen Extremfall dar: keine der 200 Szenen steht in Konflikt zu irgendeiner anderen. Die Berechnungsdauer in so einem Fall grenzt an Null.

Der gelbe und der blaue Graph zeigen Zwischenfälle, die eher erwartet werden können. Blau dargestellt sind zwei Blöcke von 100 Szenen, die jeweils in ihrem Block miteinander konfliktieren, aber nicht mit denen des anderen Blocks. Die Blocks wurden nacheinander komplett erstellt, was zeigt, dass die jeweilige Berechnungsdauer unabhängig von der Existenz und Größe anderer Blocks ist.

Die gelbe Kurve zeigt einen ähnlichen Fall mit 4 Blöcken von jeweils 50 Szenen. Allerdings wurden hier alle Blöcke zur gleichen Zeit erstellt und die Szenen gleichmäßig verteilt hinzugefügt. Deswegen ist der Graph nicht in Stufen aufgeteilt, sondern eine Gerade.

Fazit

Die Performanz des Choco Solvers reicht für die Szenarienverwaltung komplett aus. Selbst im Worst-Case beginnen problematische Zeiten erst nach deutlich über 100 Szenen aufzutreten, während im Average-Case das Bundle tausende Szenarien verwalten kann.

6.3 Quellcode

6.3.1 Vorteile

Die Implementierung wurde möglichst generisch gehalten. Dank dieses Prinzips lässt sich das Szenenmanagement in Zukunft mühelos warten. Man kann leicht neue Arten von Zielzuständen definieren - die *AbstractTargetState*-Klasse lässt sich problemlos um neue Funktionalitäten erweitern.

Die Elemente der Constraint Satisfaction sind vom restlichen Programm los gekoppelt. Falls nötig, lassen sie sich sowohl leicht erweitern, als auch komplett aus dem Programm entfernen. Vor allem das Hinzufügen von komplexen Constraints, wie in Kapitel 3 erläutert, lässt sich nahezu trivial bewerkstelligen.

Bei der Definition von Zeitintervallen ist dem Nutzer maximale Freiheit geboten. Er kann sowohl konkrete Zeiten mit beliebigen Wiederholungsintervallen definieren, als auch eines aus einer Reihe von vordefinierten Wiederholungsverhalten auswählen.

Ebenso besteht es mit dem Einstellen der Zielzustände. Der Nutzer kann innerhalb eines Szenarios für *ein* Gerät Minimum, Maximum *und* Zielwert gleichzeitig definieren.

6.3.2 Nachteile

Das einzige Problem, was nicht *komplett* behoben oder umgangen werden konnte, besteht in der Unterscheidung von manueller und automatischer Schaltung von Geräten. Obwohl die in Kapitel 4 erläuterte Funktionsweise in den meisten Fällen eindeutig zwischen den Fällen unterscheiden kann, gibt es einen Spezialfall, wo das nicht der Fall ist:

Wenn der Nutzer und ein Szenario zur *gleichen* Zeit *ein* Gerät auf den *gleichen* Zustand schalten, so kann es vorkommen, das die manuelle Schaltung des Nutzers als automatische interpretiert wird. In so einem Fall würde der Timer nicht gesetzt werden, was zu einem inkorrekten Verhalten des Systems führen würde.

6.4 Fazit

Die Implementierung erfüllt sowohl alle explizit definierten Anforderungen an die Funktionsweise, als auch die impliziten Anforderungen an die Schnelligkeit. Sie lässt sich leicht um neue Elemente und Funktionalitäten erweitern.

Kapitel 7

Zusammenfassung

7.1 Zusammenfassung

Im Rahmen der Arbeit wurden drei verschiedene Ansätze zur Konfliktresolution für Szenarien analysiert. Es hat sich herausgestellt, dass nur die Constraint Satisfaction für das Lösen des Problems von Nutzen ist. Die Konflikte werden behoben, indem Szenen Prioritäten zugewiesen bekommen, die mit Hilfe von CSPs berechnet werden.

TLA+ kam nicht zum Einsatz, da eine formale Spezifikation des Problems keine besonderen Vorteil bieten würde. Außerdem wäre der Aufwand für die Formalisierung eines komplexen Prioritätenvergabe-Verfahrens nicht vertretbar.

Ähnlich besteht es mit den Business Rules Management Systems. Es hat sich herausgestellt, dass die Komplexität und vor allem die Variabilität des Problems einen Einsatz von BRMS nicht rechtfertigt. Der entstehende Overhead durch die ständige Evaluation der Rules wird nicht durch die erhöhte Flexibilität kompensiert.

Auf Basis des gewählten Ansatzes wurde ein vollständiges Lösungskonzept entworfen und implementiert. Der Code ist möglichst generisch gehalten, sodass Erweiterungen mit geringem Aufwand eingebaut werden können.

Die Constraint Satisfaction wurde mit Hilfe von "Choco Solver 3" realisiert, einer open-source Java-Bibliothek zum Erstellen und Lösen von CSPs.

In der Evaluationsphase wurde die Implementierung auf Vorteile und Nachteile analysiert, sowie die Leistungsfähigkeit der CSP-Elemente einzeln geprüft. Es stellte sich heraus, dass die CSP-Elemente mit der erwarteten Last problemlos fertig werden. Dennoch kann es in sehr ungünstigsten Fällen zu leichten Verzögerungen kommen.

7.2 Ausblick

Da Szenarien erstellt und verwaltet werden können, stellt sich die Frage, wie das Projekt fortgesetzt werden kann. Da es unzählige mögliche Verbesserungen und Erweiterungen gibt, werden hier nur drei davon kurz vorgestellt.

7.2.1 Manuell definierte Constraints erlauben

Eine offensichtliche Fortsetzung der Arbeit wäre die Erweiterung der Szenen um vom Nutzer erstellte Constraints. Es wäre sinnvoll, dass der Nutzer, wie in Kapitel 3 vorgestellt, "beliebige" eigene Constraints definieren könnte.

Hier werden die Vorteile der aktuellen Implementierung besonders deutlich. Dank der generischen Implementierung und der Kapselung der CSP-Logik lässt sich eine derartige Erweiterung ohne Probleme realisieren.

GUI

Eine andere Möglichkeit, die Arbeit fortzuführen, wäre es eine passende GUI zu entwickeln, die einerseits das Erstellen von eigenen Constraints ermöglichen würde, andererseits dem Anspruch des Qivicon-Projekts, an ältere Menschen gerichtet zu sein, gerecht würde.

Automatische Generation von Szenarien

Eine weitere mögliche Entwicklungsrichtung ist das automatische Generieren von Szenarien. Das System könnte eine Statistik der manuellen Schaltungen des Nutzers erstellen und, basierend darauf, dem Nutzer selbst neue Szenarien vorschlagen.

Anhang A

Weitere Informationen

Der Quellcode der Implementierung befindet sich auf der mit der Arbeit abgegebenen CD.

Abbildungsverzeichnis

Literaturverzeichnis

[1] *What is Business Rules Management?* Website. Available online at http://www-01.ibm.com/software/websphere/products/business-rule-management/whatis/; visited on October 25th, 2013.

[2] *Backbone.js*, October 2013. Available online at http://backbonejs.org/; visited on October 25th, 2013.

[3] *Choco 3.* Website, 2013. Available online at http://www.emn.fr/z-info/choco-solver/index.php?page=choco-3; visited on October 25th, 2013.

[4] *Qivicon.* Website, 2013. Available online at https://www.qivicon.com/; visited on October 25th, 2013.

[5] *Qivicon Documentation.* Part of Qivicon SDK, 2013. Only available to developers.

[6] *Temporale Logik der Aktionen.* Website, August 2013. Available online at http://de.wikipedia.org/wiki/Temporale_Logik_der_Aktionen; visited on October 25th, 2013.

[7] *The OSGi Architecture.* Website, 2013. Available online at http://www.osgi.org/Technology/WhatIsOSGi; visited on October 25th, 2013.

[8] Bali, Michal: *Drools JBoss Rules 5.0 Developer's Guide.* Packt Publishing Ltd., 32 Lincoln Road Olton Birmingham, B27 6PA, UK., July 2009. Available online at https://drooledphones.googlecode.com/svn/wiki/Drools%20JBoss%20Rules%205.0%20Developer's%20Guide_1847195644.pdf; visited on October 25th, 2013.

[9] Barták, Roman: *Online Guide to Constraint Programming*, 1998. Available online at http://ktiml.mff.cuni.cz/~bartak/constraints/index.html; visited on October 25th, 2013.

[10] Bartlett, Neil: *OSGi In Practice.* Available online at http://njbartlett.name/files/osgibook_preview_20091217.pdf; visited on October 25th, 2013, December 2009.

[11] Blance, Juan Jose und Lina Khatib: *Course Scheduling as a Constraint Satisfaction Problem.* Seiten 1–8, 1998. Available online at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.8460&rep=rep1&type=pdf`; visited on October 25th, 2013.

[12] Laburthe, F. und N. Jussien.: *Choco Solver Documentation*, 2011. Available online at `http://www.dei.unipd.it/~pini/fse-doc/corsi/choco-doc.pdf`; visited on October 25th, 2013.

[13] Lamport, Leslie: *Principles and Specifications of Concurrent Systems*, July 2012. Available online at `http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html`; visited on October 25th, 2013.

[14] Lamport, Leslie: *A PlusCal User's Manual, C-Syntax Version 1.8*, February 2013. Available online at `http://research.microsoft.com/en-us/um/people/lamport/tla/c-manual.pdf`; visited on October 25th, 2013.

[15] Liu, Zhe: *Algorithms for Constraint Satisfaction Problems (CSPs).* Master of Mathematics in Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1998. Available online at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.74.4252&rep=rep1&type=pdf`; visited on October 25th, 2013.

[16] Rudolph, George: *Some Guidelines For Deciding Whether To Use A Rules Engine.* Website, July 2008. Available online at `http://www.jessrules.com/guidelines.shtml`; visited on October 25th, 2013.

[17] Runte, Wolfgang: *YACS: Ein hybrides Framework für Constraint-Solver zur Unterstützung wissensbasierter Konfigurierung.* Diplomarbeit, Universität Bremen, 1 2006. Available online at `http://www.inf.uos.de/woru/pub/diplom/html/Diplom.html`; visited on October 25th, 2013; Definition 5.2.1.

[18] Singh, Inderjeet, Joel Leitch und Jesse Wilson: *Gson User Guide*, 2008. Available online at `https://sites.google.com/site/gson/gson-user-guide`; visited on October 25th, 2013.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 18. November 2013

Konstantin Tkachuk