CARL VON OSSIETZKY universität OLDENBURG

Abteilung Umweltinformatik
Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

MASTERARBEIT

Untersuchung und Entwicklung von Verfahren zur Desynchronisation adaptiver Kühlgeräte

Christian Hinrichs

Erstgutachter Prof. Dr. Michael Sonnenschein
Zweitgutachterin Dr. Ute Vogel

Vorgelegt: 24. Oktober 2008

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Oldenburg, am 24. Oktober 2008

Christian Hinrichs

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Eine Hauptaufgabe von Energieerzeugern ist es, die Verfügbarkeit von Energie konstant und ohne Schwankungen zu decken. Im Zuge der Versorgung mit regenerativen Energien wird dies allerdings immer schwieriger, da solche Energieträger meist nicht steuerbar sind. Der Unterschied zwischen verfügbarer Energie und Verbrauch muss durch die Energieerzeuger ausgeglichen werden, um die Netzstabilität zu gewährleisten (geregelt durch die ETSO, siehe [13]). Im Falle eines Energieüberschusses besteht die Möglichkeit, einzelne Kraftwerke herunterzufahren oder die überschüssige Energie zu speichern, um sie später zu verwenden. Im umgekehrten Fall, bei einem Energiedefizit, gibt es ebenfalls zwei Möglichkeiten: eine Erhöhung der Produktion durch Hinzuschalten weiterer Kraftwerke, oder eine Reduzierung des Verbrauches durch geschicktes Lastmanagement. Diese Arbeit behandelt einen Aspekt des letzten Punktes, und zwar die Steuerung des Verbrauches von Kühlgeräten in Privathaushalten zum Zweck der Lastverschiebung.

1.1 Ziele dieser Arbeit

Die gängigen Kühlschränke arbeiten nach dem Prinzip der Kompressionskältemaschine. Sie wandeln elektrische Energie in Kälte um, indem über ein flüssiges Kühlmittel die Wärme des Kühlschrankinnenraumes nach außen transportiert und dort abgegeben wird. Dieser Kühlvorgang läuft allerdings nicht permanent ab, da die Kühlschrankinnenräume gegen ihre Umgebung isoliert sind und der Kühlmechanismus wesentlich schneller kühlt als die Kühlschränke Wärme aus ihrer Umgebung aufnehmen. Aus diesem Grund können Kühlschränke als thermische Speicher angesehen werden: sie kühlen ihren Innenraum auf eine Minimaltemperatur ab und

begeben sich dann in einen passiven Zustand. Erst wenn eine Maximaltemperatur überschritten wird, startet der Kühlvorgang erneut. Durch die variierenden Modell- sowie Nutzungscharakteristika der Kühlschränke ergibt sich bei der Betrachtung der Masse der (z. B. in Deutschland) im Betrieb befindlichen Geräte eine nahezu gleichverteilte Last über die Zeit ohne nennenswerte Lastspitzen. In Bezug auf das Lastmanagement bietet sich nun folgende Möglichkeit: ist in einem nahen Zeitraum ein Energiedefizit auf der Seite der Energieerzeuger zu erwarten, so könnte dieses mehr oder weniger durch die Verschiebung der Last der zu dieser Zeit im Aktiv-Betrieb befindlichen Kühlschränke ausgeglichen werden. Dazu gibt es bereits diverse Ansätze (siehe etwa [15, 33]), diese Arbeit bezieht sich jedoch auf eine spezielle Methode von Stadler et al. [34], nach der ein Steuerungssignal für eine Lastverschiebung mit Vorankündigung erfolgt. Die Kühlschränke nutzen die Vorlaufzeit dann dazu, ihre Temperatur nach verschiedenen vorgeschlagenen Strategien soweit abzusenken, dass sie das zu überbrückende Intervall wie gewünscht im Passiv-Betrieb verbringen können. Die Simulationen von Stadler et al. ergaben unter anderem jedoch, dass das Eingreifen in die vorher gleichverteilte Last der Kühlschränke zu einer Synchronisation der Kühlungsphasen führte, sodass zwar das Energiedefizit im Regelzeitraum mehr oder weniger ausgeglichen werden konnte, die sich danach synchron wiederholenden Kühlungsphasen jedoch diesen Vorteil durch enorme Lastspitzen wieder neutralisierten. Die Zielsetzung dieser Arbeit ist daher die Untersuchung des Reaktionsverhaltens der Kühlschränke auf die Parameter des Kontrollsignals sowie die Entwicklung eines Verfahrens zur Desynchronisation der Geräte nach einer erfolgten Kontrollphase, um die Gleichverteilung der Last wiederherzustellen.

Da sich die oben beschriebene Problemstellung auf emergentes Verhalten einer großen Menge von Kühlschränken bezieht, die einerseits stark inhomogen sind (d.h. schwankende Betriebsparameter von Gerät zu Gerät) und andererseits gewissen Zufallseinflüssen durch den Nutzer etc. unterliegen, lässt sich an dieser Stelle nur schwerlich ein analytisches Lösungsverfahren ansetzen. Daher wird der gleiche Ansatz wie in [34] verfolgt und eine Desynchronisationsstrategie mit Hilfe von Simulationen entwickelt. Aus der Menge der existierenden Techniken zur Modellbildung wird ein ereignisdiskretes individuenbasiertes Modell verwendet (*individual-based configuration model* - IBCM, vgl. [32]), da sich dieses gut für die Simulation emergenten Verhaltens eignet. Mit Hilfe der Simulationsumgebung werden dann Möglichkeiten zur Dämpfung der Synchronisation der Geräte entwickelt und deren Potenzial bei variierenden Geräte- sowie Steuerparametern untersucht. Zusätzlich wird die eigentliche Modellierung der Geräte evaluiert. Für die Bewertung der zu er-

stellenden Desynchronisationsverfahren gilt generell, dass diese sich nicht nur in ihrer Performanz, sondern auch in der Realisierbarkeit unterscheiden können. So würde ein autonomeres (und damit intelligenteres) Verhalten der Kühlschränke zusätzliche Anforderungen an die Controller der Geräte bezüglich Logik und differenzierter Kontrolle der Kühltechnik stellen, was mit zusätzlichen Kosten verbunden ist und sich daher antiproportional zur Machbarkeit verhält. Andererseits ist es fraglich, inwieweit das gewünschte Verhalten durch alleiniges Modifizieren der Parameter des Steuerungssignals erreicht werden kann, sodass hier ein akzeptabler Mittelweg gefunden werden sollte.

1.2 Aufbau

Die vorliegende Arbeit wird wie folgt gegliedert sein: Zunächst werden in Kapitel 2 verschiedene Simulationsumgebungen und -frameworks betrachtet und eines davon zur Verwendung ausgewählt. Kapitel 3 wird sich mit dem zu simulierenden Modell beschäftigen, seine Struktur und Arbeitsweise beschreiben und im Vergleich zum Basismodell aus [34] um einige Komponenten erweitern. Auch die Implementierung des Modells in dem ausgewählten Simulationsframework wird hier erläutert. Anschließend werden in Kapitel 4 die in [34] eingeführten Steuersignale dargestellt und implementiert. Nachfolgend beschreibt Kapitel 5 die Entwicklung von verschiedenen Verfahren zur Dämpfung der Synchronisation, welche dann in Kapitel 6 auf ihre Robustheit bei unterschiedlichen Geräteparametern der simulierten Kühlgeräte sowie variierenden Parametern der Steuersignale hin untersucht werden. Kapitel 7 beschäftigt sich dann mit der Umsetzung der Simulationsmodelle in konkrete Hardware-Controller. Zum Schluss werden in Kapitel 8 auf theoretischer Basis weitere Möglichkeiten zur Nutzung der vorgestellten Ansätze zur Lastreduktion betrachtet und in Kapitel 9 ein Gesamtfazit über die gewonnenen Erkenntnisse gezogen. Die Anhänge A, B und C enthalten weitere Erläuterungen und Ergebnisse, welche als Ergänzung zum Haupttext betrachtet werden sollen und an den entsprechenden Stellen referenziert werden. In Anhang E ist abschließend der Inhalt der beigefügten CDROM beschrieben.

Kapitel 2

Simulationsframeworks

Um eine Simulation eines individuenbasierten Modells durchzuführen, wird im Kontext der objektorientierten Programmierung prinzipiell nur ein zentraler Zeitgeber benötigt, nach welchem sich die als Objekte vorliegenden Entitäten des Modells (=Individuen) richten und ihre Interaktion steuern. Insbesondere bei überschaubaren Modellen mit nur wenig Wirkungsbeziehungen zwischen den einzelnen Individuenklassen würde sich ein solcher Ansatz schnell realisieren lassen. Die entstehende Simulation wäre stark auf das Modell angepasst. Im vorliegenden Fall jedoch beinhaltet die Aufgabenstellung die Erweiterung des Modells um noch zu erarbeitende Komponenten, weshalb eine stark angepasste Simulationsumgebung hier schnell unflexibel und zu starr wäre.
In dieser imaginären einfachsten Implementierung würde die Simulation zudem zeitdiskret ablaufen, d. h. zu jedem simulierten Zeitschritt würden die Zustände aller Individuen aktualisiert werden, was rechen- und daher zeitintensiv, aber leicht zu programmieren ist. Im Gegensatz dazu werden im ereignisdiskreten Ansatz nur diejenigen Zeitpunkte betrachtet, in denen sich der Zustand des Systems ändert. Zu diesen Zeitpunkten werden die Zustände derjenigen Individuen aktualisiert, die durch die zu diesem Zeitpunkt anliegenden Ereignisse beeinflusst werden. Dieser Ansatz aus Sicht einer Kette von Ereignissen ist flexibler, beinhaltet aber einen größeren Implementationsaufwand, da neben dem Zeitgeber und der Ereignisbehandlung in den Individuen auch eine intelligente zentrale Ereignisverwaltung vonnöten ist, welche die Ausführung der gesamten Simulation kontrolliert.
Aufgrund dieser Argumente wurde verstärkt nach bereits existierenden Simulationsumgebungen und -frameworks gesucht, die in Größe und Funktionsumfang für diese Arbeit geeignet sind.

2.1 Die Kandidaten

Die folgenden Abschnitte beschreiben einen Teil der Kandidaten in knapper Form. Sie richten sich alle nach dem ereignisdiskreten Ansatz, da dieser die Oberklasse darstellt und durch passende Modellierung leicht eine zeitdiskrete Simulation erreicht werden kann. Anschließend wird ein Fazit über die erkundeten Frameworks gezogen und eines davon zur Verwendung in dieser Arbeit ausgewählt.

2.1.1 Hobo

Hobo[1] wurde 1992-1993 von Ladislav Lhotka in der Programmiersprache Smalltalk entwickelt und besteht aus einer Sammlung von frei verfügbaren Bibliotheken zur Entwicklung und Simulation individuenorientierter Modelle mit verschiedenen Individuenklassen in wahlweise komplexer räumlicher Verteilung.
Der Autor nennt drei Hauptaspekte der Software (vgl. [27]):

1. Räumlichkeit
   (inhomogene Verteilung der Individuen über die simulierte Umgebung),
2. Aggregation
   (Individuen können in hierarchische Klassen eingeteilt werden) und
3. Verhaltensweisen
   (algorithmische Beschreibung von Entscheidungprozessen).

Hobo ist eine frühe Entwicklung im Bereich der Frameworks für individuenorientierte Simulation. Bis vor einigen Jahren war zur Verwendung eine Smalltalk-Lizenz notwendig, dies hat sich mit der freien Distribution Squeak[2] jedoch erübrigt.

2.1.2 SimPy

SimPy[3] steht für *Simulation in Python* und ist ein freies Simulationsframework, entwickelt von Klaus G. Müller und Tony (G. A.) Vignaux. Es existiert seit 2002 und ist in der Sprache Python geschrieben. Das Paket enthält neben dem eigentlichen Simulationskern einige Komponenten, die den Nutzer bei der Ausführung der Simulationen sowie Visualisierung der Ergebnisse unterstützen. Zur Zeit liegt SimPy

[1] http://staff.cesnet.cz/~lhotka (letzter Zugriff: 24. Oktober 2008)
[2] http://www.squeak.org (letzter Zugriff: 24. Oktober 2008)
[3] http://simpy.sourceforge.net (letzter Zugriff: 24. Oktober 2008)

in der Version 1.9.1 vor, ausgelegt auf Python 2.3. Die Software modelliert Systeme auf der Basis von Prozessen, wobei jede aktive Entität als Prozess vorliegt. Zudem existieren verschiedene Ressourcenklassen, um unterschiedliche Arten von Entitäten- und Ressourcentransfers modellieren zu können, wie etwa kontinuierlichen Rohstoffverbrauch (mittels der Klasse *Level*) oder die Inanspruchnahme von limitierten Ressourceneinheiten (über die Klasse *Resource*). Siehe dazu auch [37].

SimPy wird permanent weiterentwickelt und erfreut sich großer Beliebtheit in der Python-Gemeinschaft.

2.1.3 JSIM

Das Simulationsframework JSIM[4] entstand um 1998 am Computer Science Department, University of Georgia unter der Aufsicht von John E. Miller. Es ist in Java geschrieben und verfolgt den Ansatz, die Teile eines Modells in Java Beans zu kapseln. Die Autoren versuchen so, eine komponentenbasierte Software zu schaffen, welche modular und flexibel genug ist um etwa auch als dynamische Webanwendung agieren zu können (vgl. [29]). Die Modelle in JSIM folgen ebenfalls dem Paradigma der prozessorientierten Modellierung. Eine Besonderheit des Frameworks ist die Möglichkeit der Simulation in „Pseudo-Realzeit“, welche eine animierte Liveansicht des simulierten Modells erlaubt. Außerdem enthält die Software das Paket JMODEL, das die grafische Erstellung von Modellen für JSIM erlaubt.

2.1.4 simjava

Die Autoren F. Howell und R. McNab, Department of Computer Science, University of Edinburgh, betrachten simjava[5] als Toolkit zur Erstellung von Modellen komplexer Systeme im ereignisdiskreten Kontext [21]. Die Software ist eine Neuimplementierung in Java des Simulationsframeworks HASE/HASE++, welches in C++ geschrieben wurde und bereits viele Funktionalitäten wie grafische Editoren, statistische Analysen, Animationen sowie persistente Datenhaltung bietet (siehe etwa [22]). simjava zielt ähnlich wie JSIM darauf ab, die Simulationen in Webseiten einzubetten und in einer Liveansicht darzustellen. Die aktuellste Version des Frameworks ist 1.2 beta von 1997.

[4] http://www.physiome.org/jsim (letzter Zugriff: 24. Oktober 2008)
[5] http://www.dcs.ed.ac.uk/home/hase/simjava (letzter Zugriff: 24. Oktober 2008)

2.1.5 JavaSim

JavaSim[6] wurde zwischen 1994 und 1998 von M. C. Little am Department of Computing Science, Computing Laboratory, University of Newcastle upon Tyne entwickelt und ist eine Java-Neuimplementierung des Frameworks C++SIM, welches ursprünglich aus dem weiteren Umfeld des Projektes Arjuna hervorging. Arjuna stellt ein Framework zur Entwicklung fehlertolerierender, verteilter Systeme in C++ dar und wurde um 1991 entwickelt (siehe [31]). JavaSim wird von den Autoren als direkter Nachfolger dieser Systeme beschrieben und erbt demzufolge viele ihrer Eigenschaften. Es verwendet ebenfalls die Modellierung auf der Basis von Prozessen und nutzt dabei direkt die Konzepte der Simulationssprache SIMULA (siehe [28]), welche in den 1960er Jahren zur Simulation von physikalischen Prozessen entstand. JavaSim bietet keine Visualisierung des Modellierungsprozesses oder der Simulationsergebnisse. Die Software liegt in der Version 0.3 vor und scheint seit 1998 nicht weiter entwickelt worden zu sein.

2.1.6 javaSimulation und jDisco

javaSimulation[7] basiert wie JavaSim auf den Konzepten aus SIMULA. Es wurde um 2000 unter der Leitung von K. Helsgaun am Department of Computer Science, Roskilde University entwickelt. Das Paket ist wie die vorhergehenden in Java geschrieben und ist für die Modellierung von Systemen aus der prozessorientierten Sicht gedacht, unterstützt aber auch die ereignis- und aktivitätsorientierte Sicht (siehe [19]). Um eine unnötige Menge an einzelnen Java Threads zu vermeiden, setzt die Software das Konzept der Coroutinen aus SIMULA um, welches Pseudoparallelität durch wechselseitige Ausführung von Prozeduren ermöglicht, was im Prinzip etwa einer Sequentialisierung unter Ausschluss der Interferenz entspricht (vgl. [1]). Während javaSimulation auf rein diskreter Basis arbeitet, bietet die Erweiterung jDisco[8] der gleichen Autoren die Möglichkeit, diskrete und kontinuierliche Modelle in Kombination miteinander zu simulieren [20]. Beide Frameworks liegen zur Zeit in der Version 1.1 von 2004 vor.

---

[6] http://javasim.ncl.ac.uk (letzter Zugriff: 24. Oktober 2008)

[7] http://www.akira.ruc.dk/~keld/research/JAVASIMULATION (letzter Zugriff: 24. Oktober 2008)

[8] http://www.akira.ruc.dk/~keld/research/JDISCO (letzter Zugriff: 24. Oktober 2008)

2.1.7 J-Sim

J-Sim[9] wurde seit 2000 an der Faculty of Applied Sciences, University of West Bohemia entwickelt und liegt aktuell in der Version 0.6.0 von 2006 vor. Das Java Framework ist hauptsächlich zur Modellierung von Warteschlangennetzwerken (*queueing networks*, siehe [10]) gedacht, kann aber auch für andere Modelle verwendet werden. Auch dieses Framework hält sich stark an SIMULA, ist genaugenommen aber eine Neuimplementierung des prozedural in ANSI C programmierten Frameworks C-Sim (siehe [23, S.14]). Im Gegensatz zu javaSimulation verwendet J-Sim bewusst Java Threads, um quasiparallele Prozesse zu simulieren. Das Framework enthält vorbereitete Klassen zur Visualisierung von Modellen.

2.1.8 DESMO-J

DESMO-J[10] (*Discrete Event Simulation and MOdelling in Java*) wurde von der Arbeitsgruppe Angewandte und Sozialorientierte Informatik des Department Informatik an der Universität Hamburg unter der Leitung von B. Page entwickelt und liegt zur Zeit in der Version 2.1.2, März 2008 vor. Es enstand aus dem in Modula-2 geschriebenen Framework DESMO, welches 1989 ebenfalls an der Universität Hamburg entwickelt wurde und seinerseits Konzepte aus dem 1979 von Graham Birtwistle geschriebenen Paket DEMOS umsetzt. DESMO wurde bereits in mehrere Sprachen portiert, darunter SmallTalk, C++ und Delphi (noch in Arbeit) [14]. Die grundlegenden Komponenten der Java Version DESMO-J wurden 1999 geschrieben, seitdem entwickelt die Arbeitsgruppe von Prof. Page in Kooperation mit Prof. W. Kreutzer vom Department of Computer Science der University of Canterbury, Neuseeland permanent Erweiterungen zu dem Framework, darunter etwa spezielle Komponenten für die Simulation von Produktionssystemen sowie Hafenanlagen, oder ein Zusatzframework zur Optimierung verteilter Simulationen (DISMO). Aktuelle Arbeiten an DESMO-J umfassen beispielsweise die Integration in die Entwicklungsumgebung Eclipse, um insbesondere komplexe Modelle komfortabel verwalten, visualisieren und auswerten zu können. Bemerkenswert an DESMO-J ist außerdem, dass das Framework ähnlich wie javaSimulation sowohl die prozessorientierte als auch die ereignisorientierte Modellierung unterstützt.

[9] http://www.j-sim.zcu.cz (letzter Zugriff: 24. Oktober 2008)
[10] http://asi-www.informatik.uni-hamburg.de/desmoj (letzter Zugriff: 24. Oktober 2008)

2.1.9 SimKit

SimKit[11] wurde zuerst 1996 von K. Stork in einer Masterarbeit an der Naval Postgraduate School, Monterey, California unter der Betreuung von Prof. A. H. Buss (MOVES Institute) entwickelt [35] und anschließend von Prof. Buss fortgeführt. Das ursprünglich aus Gründen der webbasierten Portabilität in Java geschriebene Framework liegt zur Zeit in der Version 1.3.4 vom Januar 2008 vor. Ebenso wie DESMO-J unterstützt SimKit die prozess- und ereignisorientierte Modellierung [4], der Autor empfiehlt jedoch ausdrücklich die ereignisorientierte Modellierung anhand der Technik des *Event Graph Modeling*, einer grafischen Beschreibung der Modellkomponenten und ihrer Ereignisse (siehe [5, 6]). Neuere Entwicklungen um SimKit betreffen insbesondere das Design komplexer Modelle andhand von komponentenbasierten Techniken, welche das recht einfach gehaltene Framework SimKit um die Möglichkeit der Modellierung von sehr komplexen Systemen erweitern. Erreicht wird dies durch Listener-Konzepte zur Vernetzung modularer Teilmodelle (siehe [7]). Parallel dazu wird aktuell das Tool Viskit[12] entwickelt, welches die grafische Modellierung anhand des *Event Graph Modeling* unterstützt und SimKit als Simulationsengine enthält (siehe [8]).
Die Besonderheit an SimKit ist, dass die Software im Gegensatz zu den bisher vorgestellten Frameworks bereits eine interne statistische Verarbeitung der Zustandsvariablen der Modelle enthält. Ermöglicht wird dies durch Konzepte auf Basis standardisierter *PropertyChangeListener*.

2.1.10 JAPROSIM

JAPROSIM[13] wurde an der University of Batna von A. Bourouis und B. Belattar entwickelt, die aktuellste Version ist das Release „BlueBird“ vom Januar 2007. Das Framework ist in Java geschrieben und bietet ähnliche Funktionalitäten wie die vorhergehenden. So unterstützt es beispielsweise die prozessorientierte Sicht und enthält ein Paket zur grafischen Bearbeitung der Modelle sowie zur Visualisierung der Simulationen. Zusätzlich ist wie bei SimKit ein Mechanismus zur automatischen Ansammlung von statistischen Informationen bezüglich der Zustandsvariablen des Modells enthalten (siehe [3]). JAPROSIM verwendet ähnlich wie J-Sim intensiv Java

[11] http://diana.cs.nps.navy.mil/simkit (letzter Zugriff: 24. Oktober 2008)
[12] http://diana.cs.nps.navy.mil/Viskit (letzter Zugriff: 24. Oktober 2008)
[13] http://sourceforge.net/projects/japrosim (letzter Zugriff: 24. Oktober 2008)

Threads, um die einzelnen Prozesse des Modells zu simulieren.

2.2 Auswahl eines Frameworks

Die in den vorherigen Abschnitten beschriebenen Projekte stellen nur eine Auswahl an Frameworks für die individuenorientierte diskrete Simulation dar. Der Hauptfokus wurde auf Java[14] Frameworks gelegt, es existieren jedoch wie in den einzelnen Beschreibungen bereits angedeutet auch viele Bibliotheken in anderen Sprachen wie SmallTalk, C, C++, Python, Delphi und weitere. Im Vergleich zu den meisten dieser Sprachen bietet Java jedoch einige Vorteile. So wird die Sprache etwa vermehrt im forschungsorientierten Umfeld verwendet und kann durchaus als sehr aktuelle Programmiersprache gesehen werden, dies spiegelt sich in den Releasedaten der oben beschriebenen Projekte wider. Zudem ist Java äußerst portabel, die Vorteile der plattformunabhängigen Programmierung liegen hier auf der Hand. Weiterhin ist Java durch die native Unterstützung von Threads gut für die (quasi-)parallele Programmierung geeignet. Das umfangreiche Angebot an mitgelieferten Klassen ermöglicht es außerdem, schnell und leicht weitere Elemente wie grafische Benutzungsoberflächen, Visualisierungen, Netzwerkunterstützung (z. B. für verteilte Programme) oder ähnliches zu implementieren, wovon einige der oben genannten Frameworks intensiv Gebrauch machen.

Es gestaltet sich recht schwierig, aus den vorgestellten Frameworks einen einzelnen Kandidaten für diese Arbeit auszuwählen, da jedes seine eigenen Vor- und Nachteile hat. Betrachtet man die vorliegende Aufgabenstellung, so lassen sich allerdings einige Kriterien herausfiltern, nach denen die einzelnen Kandidaten bewertet werden können:

- **Simplizität**: Das zu modellierende System der Kühlschränke umfasst keine komplexe hierarchische Individuenstruktur. Außerdem ist es nicht notwendig, örtliche Abhängigkeiten und Interaktionen zu modellieren. Das entstehende Modell wird überschaubar sein.
- **Auswertbarkeit**: Die Analyse einzelner Parameter sowie die Entwicklung verschiedener Verfahren, um ein gewünschtes Verhalten zu erreichen, erfordert ei-

[14] http://java.sun.com (letzter Zugriff: 24. Oktober 2008)

ne möglichst gute Auswertung der Simulationsparameter und ihrer Ergebnisse. Dazu zählt auch die Aggregation statistischer Rohdaten.

- **Handhabung**: Die Erstellung des Modells und seiner Simulation liegt nicht im Hauptfokus dieser Arbeit, daher ist die Verwendung eines gut dokumentierten und einfach zu handhabenden Frameworks sinnvoll. Insbesondere Java Frameworks bieten sich hier an, da ich im Vorfeld dieser Arbeit bereits umfassende Erfahrungen mit dieser Programmiersprache gesammelt habe, was der Effizienz zugute kommt.

- **Einsatzgebiet**: Da die Simulation hier ein Hilfsmittel darstellt, kann auf erweiterte Funktionalitäten wie Webfähigkeit, animierte Live-Ansichten, Datenbankanbindung oder ähnliches verzichtet werden. Augenmerk ist hier eher auf die Performance zu legen, da die zu simulierende Menge an Individuen unter Umständen recht groß werden kann und schnellere Simulationsergebnisse von größerem Nutzen sind.

Unter diesen Voraussetzungen kann etwa Hobo sofort ausgeschlossen werden, da dieses Framework keines der Kriterien optimal erfüllt. SimPy wäre ein passender Kandidat, wenn es in Java geschrieben wäre. JSIM und simJava zielen laut Beschreibung der Autoren auf die Einbettung in Webseiten und die animierte Darstellung der Simulationen ab, was nicht unbedingt ein Nachteil für die vorliegende Arbeit sein muss. Jedoch würde ich diese beiden Frameworks dennoch ausschließen, da sie die anderen Kriterien nicht ausdrücklich erfüllen. JavaSim ist als Kandidat nicht geeignet, da es nach meinen Recherchen nicht komplettiert wurde und die Entwicklung 1998 mit der Version 0.3 zum Stillstand kam. J-Sim und JAPROSIM verwenden für jedes modellierte Individuum einen Java Thread, was bei großen Populationen schnell an die Grenzen der Ausführbarkeit stoßen kann.
Die verbleibenden Frameworks javaSimulation, DESMO-J und SimKit erfüllen alle drei ausreichend die obigen Kriterien. Sie sind genügend aktuell, besitzen im Falle von DESMO-J und SimKit eine beeindruckende Entwicklungsgeschichte und bieten alle die Möglichkeit, Modelle auf verschiedenen Paradigmen basierend aufzubauen.

Meine Entscheidung für diese Arbeit fällt hier auf das Framework SimKit, da es aus dieser letzten Auswahl als einziges eine vorbereitete statistische Auswertung der Simulation integriert. Mit dem Zusatzframework Viskit bietet es zudem die Möglichkeit der grafischen Modellierung. SimKit enthält zwar nur rudimentäre Möglichkeiten der Visualisierung von Simulationsergebnissen über die Grafikbibliothek AWT,

jedoch können elegantere grafische Auswertungen etwa durch die freie Bibliothek JFreeChart[15] leicht hinzugefügt werden.

[15] http://www.jfree.org/jfreechart (letzter Zugriff: 24. Oktober 2008)

Kapitel 3

Das Modell

Dieses Kapitel beschreibt das der Simulation zugrundeliegende Modell sowie seine Umsetzung im Simulationsframework SimKit. Es werden zunächst die Grundlagen der verwendeten grafischen Beschreibungssprache erläutert, anschließend folgt das Kühlschrankmodell in verschiedenen Varianten.

3.1 Event Graph Modeling

Wie bereits in Abschnitt 2.1.9 angedeutet unterstützt SimKit sowohl die prozess- als auch die ereignisorientierte Modellierung. Den Empfehlungen des Framework-Autors A. Buss folgend werden die Modelle in der vorliegenden Arbeit unter dem Paradigma der ereignisorientierten Modellierung entwickelt und betrachtet. Dieses Paradigma beschreibt ein Modell nicht aus Sicht seiner Komponenten, Zustände oder Prozesse, sondern allein aus der Sicht der auftretenden Ereignisse und deren Spezifikationen und Abhängigkeiten. Mit Hilfe von sog. „LEGOs“ (siehe [9]) ist es zwar dennoch möglich, Komponenten als einzelne Modelle zu definieren und diese dann über ein Listener Pattern miteinander zu verbinden, dies wird jedoch in der vorliegenden Arbeit nicht verwendet. Der Grund dafür liegt in der Möglichkeit der eleganteren Umsetzung der Varianten eines integrierten Modells unter Verwendung von programmierspezifischen Techniken wie Vererbung und Polymorphie.
Um die Modelle in dem ausgewählten Paradigma sinnvoll zu beschreiben, wird die von Buss vorgeschlagene Technik des *Event Graph Modeling* verwendet [5, 6]. Diese grafische Beschreibungssprache geht auf Lee Schruben zurück, der sie erstmals 1983 in [30] vorstellte. Sie definiert ein Modell über einen Graphen nach einem minimalistischen Prinzip: die einzigen vorkommenden Elemente sind Knoten, welche

Ereignisse repräsentieren, und Kanten, die das Einplanen bzw. Löschen von Ereignissen aus der Simulation darstellen, sowie beschreibende Attribute zu den Knoten und Kanten. Somit visualisiert ein derartiger Graph direkt den zeitlichen Verlauf der Ereignisliste (*Future Event List, FEL*), welche den vorgestellten Implementierungen von ereignisdiskreten Simulationen (darunter auch SimKit) als Basis dient. Um diese Zusammenhänge deutlich zu machen wird im Folgenden das Beispiel einer Schaltung für die Beleuchtung eines Kühlschrankes beschrieben. Dazu ist in Abbildung 3.1 zunächst der konventionelle Zustandsgraph einer solchen Schaltung dargestellt.

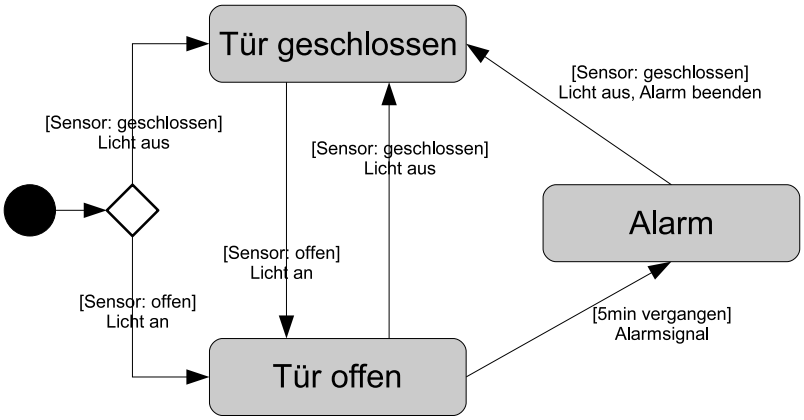


Abbildung 3.1: Ein konventionelles Zustandsdiagramm.

Diese imaginäre Schaltung besitzt einen Sensor, mit dem sie überprüfen kann, ob die Kühlschranktür offen oder geschlossen ist. Zudem existiert eine interne Zustandsvariable *state*, die den zuletzt bekannten Zustand der Tür repräsentiert. Die Schaltung soll nun so arbeiten, dass der Sensor in definierbaren Zeitintervallen abgefragt wird, ob die Tür offen oder geschlossen ist. Ändert sich durch die Meldung des Sensors der Zustand des Systems (weicht der Sensor also von der Zustandsvariablen *state* ab), so wird eine entsprechende Aktion durchgeführt. Bei einer Öffnung wird die Beleuchtung eingeschaltet und gleichzeitig ein Alarmzähler gestartet, der nach 5 Minuten ein Warnsignal gibt, wenn die Tür dann immer noch offen ist. Bei einer Schließung hingegen wird die Beleuchtung ausgeschaltet und der Alarmzähler sowie ein eventuell bereits vorhandenes akustisches Warnsignal deaktiviert. In obigem Zustandsdiagramm wurde die Schaltung mit Hilfe von drei Zuständen modelliert. Die interne Zustandsvariable *state* ist hier nicht sichtbar, sie wird implizit durch die

Zustandsknoten ausgedrückt.

Abbildung 3.2 zeigt nun die Funktionsweise der gleichen Schaltung anhand eines Ereignisgraphen nach dem Prinzip des *Event Graph Modeling*. Die Knoten in der Ab-

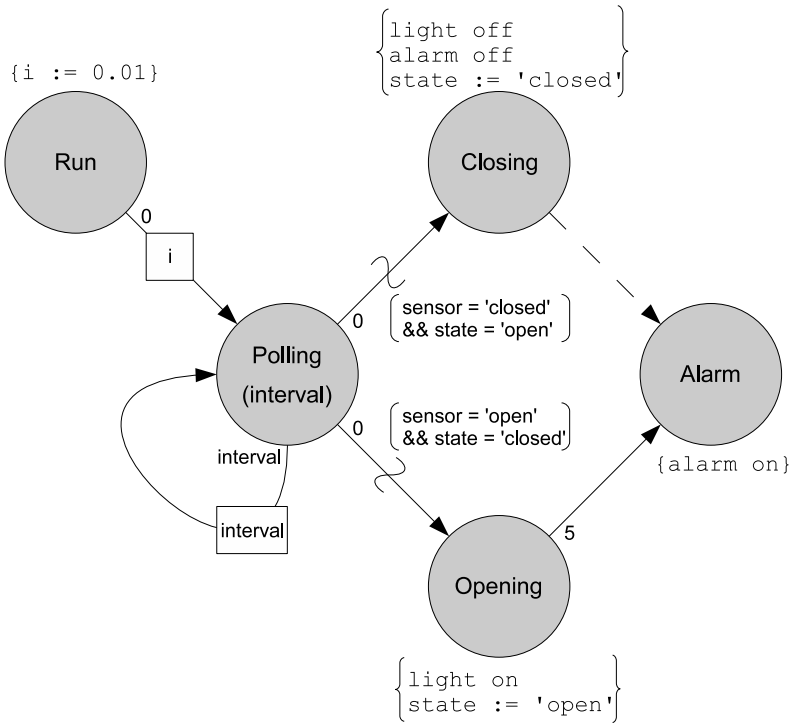


Abbildung 3.2: Exemplarischer Ereignisgraph.

bildung sind mit dem Namen des Ereignisses beschriftet, das sie repräsentieren. Steht unter dem Namen ein Wert in Klammern, so bezeichnet dieser einen Parameter, der dem Ereignis bei Eintritt übergeben wird. Die geschweiften Klammern enthalten die Operationen, die in dem jeweiligen Ereignis ausgeführt werden. Diese Operationen verändern somit den Zustand des Systems, sobald das Ereignis eintritt. Die Kanten zwichen den Ereignissen sind an ihrem Ursprungsereignis mit einer Zeitangabe versehen, die angibt, nach welchem Zeitintervall nach Eintritt des Ursprungsereignisses das Zielereignis in die Ereignisliste eingeplant wird. Parameter für das Zielereignis werden durch auf der Kante liegende eingerahmte Werte visualisiert. Soll ein Ereignis nur unter einer bestimmten Bedingung eingeplant werden, so wird die Kante mit einem Schlangensymbol sowie der entsprechenden Bedingung versehen. Es wird

außerdem zwischen durchgezogenen und gestrichelten Kanten unterschieden. Erstere planen das Zielergebnis ein, letztere löschen die nächste Einplanung des Zielereignis aus der Ereignisliste, falls vorhanden. Löschende Kanten werden grundsätzlich unmittelbar ausgeführt und daher wird ihnen kein Zeitintervall zugeordnet. Auch Parameter machen hier keinen Sinn – Bedingungen jedoch sind möglich.
Im Folgenden wird die Semantik des Ereignisgraphen dieser Schaltung genauer beschrieben. Die Beschreibung ist hier allerdings teilweise bereits auf das Framework SimKit zugeschnitten und nicht für alle Frameworks gültig.
Wird das visualierte Modell mit SimKit simuliert, so „startet“ es automatisch mit dem Ereignis Run. Dies ist eine Konvention des Frameworks. Das Ereignis dient dazu, das Modell zu initialisieren und mindestens ein Folgeereignis einzuplanen (ansonsten wäre die Simulation an dieser Stelle bereits beendet). In vorliegendem Fall initialisiert das Ereignis eine Variable i mit 0.01 und plant das Ereignis Polling nach 0 Zeiteinheiten ein. Der Einplanung wird die Variable i als Parameter übergeben. Damit ist das initiale Ereignis beendet und das Framework schreitet zum nächsten Ereignis in der Ereignisliste. Da es sich um eine ereignisdiskrete Simulation handelt, wird die Simulationszeit direkt auf die dem als nächstes eingeplanten Ereignis zugeordnete Zeit gesetzt. In diesem Fall bleibt die Simulation im Zeitpunkt 0, da Polling von Run aus mit 0 Zeiteinheiten Verzögerung, also sofort eingeplant wurde. Das nun eintretende Ereignis Polling erhält zunächst den ihm übergebenen Parameter und identifiziert ihn intern mit dem Namen interval. Da das Ereignis keine auszuführenden Operationen besitzt, werden sogleich die ausgehenden Kanten des Graphen ausgewertet und die entsprechenden Folgeereignisse eingeplant. In diesem Fall wird entweder das Ereignis Closing oder das Ereignis Opening ohne Verzögerung eingeplant, je nachdem welche von den Bedingungen an den Kanten wahr ist. Anschließend plant sich Polling selbst wieder ein, um nach einer Verzögerung von *interval* Zeiteinheiten erneut den Sensor abzufragen und eventuelle Folgeereignisse einzuplanen. Nach dieser Einplanung ist diese Instanz des Ereignisses beendet und die Simulation fährt mit dem nächsten eingeplanten Ereignis fort. War etwa die Bedingung der Kante zum Ereignis Opening wahr, so bedeutet dies, dass die Kühlschranktür geöffnet wurde. Das Ereignis Opening schaltet nun zunächst die Beleuchtung ein und aktualisiert den internen Zustand des Systems durch die Operation $state := open$. Danach wird durch Einplanung des Ereignisses Alarm mit einer Verzögerung von 5 Zeiteinheiten der Alarmzähler gestartet. Ist das Eregnis nach 5 simulierten Zeiteinheiten immer noch in der Ereignisliste enthalten, so tritt das Ereignis ein und das Warnsignal wird ausgegeben. Es kann aber auch sein, dass

während dieser Zeitspanne die Tür wieder geschlossen wird. Das Ereignis Polling, welches fortlaufend in Abständen von *interval* Zeiteinheiten den Sensor abfragt, erkennt dies, und plant das Ereignis Closing ein. Dieses Ereignis wiederum aktualisiert den internen Zustand des Systems, schaltet die Beleuchtung aus und löscht die nächste Einplanung des Ereignisses Alarm aus der Ereignisliste, und deaktiviert damit abstrakt gesehen den Alarmzähler der Schaltung.

Das minimalistische Prinzip der Ereignisgraphen ist einfach zu verstehen und gleichzeitig sehr mächtig, jedoch können einige Zusammenhänge so nicht modelliert werden. Dazu zählt etwa die Priorisierung von Ereignissen, um Konflikte bei mehrfach in einem Zeitschritt eingeplanten Ereignissen zu lösen. Außerdem ist es nicht möglich, mit der löschenden Kante *alle* folgenden Einplanungen eines Ereignisses zu entfernen, es wird immer nur das nächstgelegene gelöscht. Diese und weitere Funktionalitäten sind jedoch in SimKit implementiert und einfach nutzbar.

3.2 SimKit im Detail

Dieser Abschnitt soll eine kurze Einführung in das Simulationsframework SimKit geben. Gleichzeitig sei allerdings auf die Einführung [6] von A. Buss verwiesen, die einen guten Einblick in das Thema bietet.
Simulierbare Entitäten werden in SimKit durch das Interface SimEntity realisiert. Dieses schreibt Methoden zur Abarbeitung sowie zur Einplanung und Löschung von Ereignissen vor. Da es von weiteren Interfaces erbt und diese somit in sich vereint, definiert es zusätzlich Methoden zur Implementierung des internen Observer Patterns, welches der Aggregation statistischer Daten dient. Daneben existiert eine abstrakte Basisklasse SimEntityBase, welche das vorhergehende Interface implementiert und alle Forderungen des Interfaces erfüllt. Ereignisse werden in SimKit ähnlich wie in anderen Frameworks ebenfalls durch spezielle Objekte repräsentiert, diese stellen jedoch nur Metainformation über das Ereignis in Form des Ereignisnamens sowie weiteren Informationen wie Parametern, auslösende Entität, Zeitstempel usw. zur Verfügung. Wird solch ein Ereignis durch das Framework an die Ziel-Entität gesendet, so wird in dieser Entität mittels *Java Reflection* eine Methode aufgerufen, die dem Namen des Ereignisses plus dem Präfix „do“ entspricht. Die Aufgabe von Entwicklern ist es also lediglich, für jedes definierte Ereignis einer Entität eine entsprechende Methode mit dem Namen des Ereignisses mit Präfix „do“ zu implementieren. Der Inhalt der Methode spiegelt dann die auszuführenden Operationen dieses Ereig-

nisses wieder. Außerdem können hier etwa mittels der Befehle waitDelay(String eventName, Double delay) und interrupt(String eventName) weitere Ereignisse eingeplant oder gelöscht werden. Der Entwickler kann also den zuvor erarbeiteten Ereignisgraphen direkt in ein simulierbares Modell übersetzen. Dabei ist es empfehlenswert, die Ereignisse einer Entität über extern lesbare statische Zeichenkettenfelder bekannt zu machen, um einerseits Fehlern vorzubeugen, und andererseits die unnötige Erstellung von String-Objekten zu vermeiden. Im Verlauf dieser Arbeit werden alle Entitäten nach diesem Schema aufgebaut sein.

Sollte es notwendig sein, kann der Entwickler das Interface SimEntity auch selbst implementieren oder die abstrakte Klasse BasicSimEntity erweitern. Dies ist zum Beispiel bei umfangreichen Modellen oder langen Testreihen sinnvoll, da das in der Standardimplementierung SimEntityBase verwendete *Java Reflection* vergleichsweise langsam arbeitet und eine Anpassung auf die spezifischen Modellanforderungen deutliche Beschleunigungen ermöglicht (A. Buss hat in der Quellcode Dokumentation darauf hingewiesen, dass seine kurzen Tests ergaben, dass bei der Vermeidung von *Java Reflection* ein Geschwindigkeitsvorteil von etwa 100% erreicht wird). Diese Methodik ist in [8, Abs.2-2] beschrieben und wurde zu Testzwecken in dieser Arbeit umgesetzt. Allerdings ergaben sich mit den in den folgenden Kapiteln vorgestellten Modellen nur marginale Geschwindigkeitsverbesserungen (bei einer Population von 5000 Entitäten etwa 5%). Auch der Speicherverbrauch hat sich nicht signifikant verändert (ca. -0.17%). Diese Ergebnisse lassen sich möglicherweise durch den stark verbesserten Compiler der verwendeten Java Version 6 erklären. Daher wird im Folgenden die Standardimplementierung SimEntityBase verwendet, da sie im Vergleich zu einer eigenen Implementierung weniger fehleranfällig und leichter wartbar ist.

3.3 Das iterative Kühlschrankmodell

Das nachfolgende mathematische Kühlschrankmodell ist aus Stadler et al. [34] entnommen und basiert auf dem von Constantopoulos et al. in [11] vorgestellten Modell für Klimaanlagen und Heizungen. In diesem Modell ist die Innentemperatur T eines Kühlgerätes zu einem bestimmten Zeitpunkt t_i rekursiv zu berechnen mit

$$T_i = \varepsilon \cdot T_{i-1} + (1-\varepsilon) \cdot \left(T^O - \eta \cdot \frac{q_{i-1}}{A}\right) \quad \text{mit} \quad \varepsilon = e^{-\frac{\tau}{60} \cdot \frac{A}{m_c}} . \tag{3.1}$$

Dabei ist T_{i-1} die Innentemperatur des Gerätes zum vorherigen Zeitpunkt t_{i-1}, während T^O die (konstante) Außentemperatur beschreibt. ε definiert die Trägheit des

Systems und ist abhängig von der Isolierung A, der im Kühlschrankinneren enthaltenen thermischen Masse m_c sowie dem Parameter τ, der die Zeitspanne zwischen den Zeitpunkten t_{i-1} und t_i angibt. Im Gegensatz zu [34] und [11] wurde τ hier umgeformt, um in Minuten gemessen werden zu können, was eine Erleichterung hinsichtlich weiterer Berechnungen im Verlauf dieser Arbeit bewirkt. Der Koeffizient η beschreibt die Effizienz bzw. Performanz des Gerätes, und q_{i-1} gibt die elektrische Leistung (in Watt) an, die während des vergangenen Zeitintervalls von dem Gerät aufgenommen wurde.
Da der Parameter τ die Zeitspanne zwischen zwei Berechnungen dieser Gleichung angibt, ließe sich das Verhaltens eines Modell nach dieser Gleichung entgegen dem in der vorliegenden Arbeit verfolgten ereignisorientierten Prinzip eher mittels einer zeitdiskreten Simulation zeigen, wobei τ genau dem Takt der Simulation entsprechen würde. Da zeitdiskrete Simulationen aber eine Teilmenge der ereignisdiskreten Simulationen sind, lässt sich durch entsprechende getaktete Ereignisse auch diese Form der Simulation recht einfach in SimKit realisieren.

Legt man nun noch eine Minimal- und Maximaltemperatur T_{min}, T_{max} sowie Leistungen $q_{cooling}, q_{warming}$ für Kühl- und Aufwärmphasen und eine Zustandsvariable *state* zur Unterscheidung der aktuellen Phase (*Kühlen* oder *Aufwärmen*) fest, so ergibt sich ein charakteristisches Verhalten, nach welchem das Kühlgerät in der Phase *Kühlen* bis zu seiner Minimaltemperatur herunterkühlt, dann in die Phase *Aufwärmen* schaltet und sich langsam bis zu der Maximaltemperatur aufwärmt, um dann nach einem weiteren Phasenwechsel wieder herunterzukühlen usw. Dieses Verhalten wird im Laufe der Arbeit unter dem Begriff *reguläres Kühlprogramm* referenziert. Später folgen Strategien, die dieses Kühlprogramm modifizieren.
Tabelle 3.1 zeigt die aus [34] entnommenen Werte für obige Parameter.

Formt man Formel 3.1 um, so erhält man eine Berechnungsvorschrift für die Zeitspanne τ_{req}, die unter einer gegebenen Last Q benötigt wird, um von einer Starttemperatur T_{from} eine Zieltemperatur T_{dest} zu erreichen (in abgewandelter Form entnommen aus [34, Abs.8]):

$$\tau_{req}\left(T_{from}, T_{dest}, Q\right) \quad = \quad -\ln\left(\frac{T_{dest} - T^O + \eta\frac{Q}{A}}{T_{from} - T^O + \eta\frac{Q}{A}}\right) \cdot \frac{m_c}{A} \cdot 60\,. \tag{3.2}$$

Daraus herleiten lassen sich die Berechnungsvorschriften für die Zeitspannen $\tau_{cooling}$ und $\tau_{warming}$, die das Gerät in den Phasen *Kühlen* und *Aufwärmen* verbringt, wenn

Parameter	*Standardwert*
T^O	20.0°C
η	3.0
A	3.21
m_c	19.95 $\frac{kWh}{°C}$
τ	1 min
T_{min}	3.0°C
T_{max}	8.0°C
$q_{cooling}$	70.0 W
$q_{warming}$	0.0 W

Tabelle 3.1: Werte des Kühlschrankmodells, aus [34].

es unter einer gegebenen konstanten Last Q von T_{max} nach T_{min} kühlt bzw. sich von T_{min} nach T_{max} aufwärmt:

$$\tau_{cooling}(Q) = -ln\left(\frac{T_{min} - T^O + \eta\frac{Q}{A}}{T_{max} - T^O + \eta\frac{Q}{A}}\right) \cdot \frac{m_c}{A} \cdot 60\,, \tag{3.3}$$

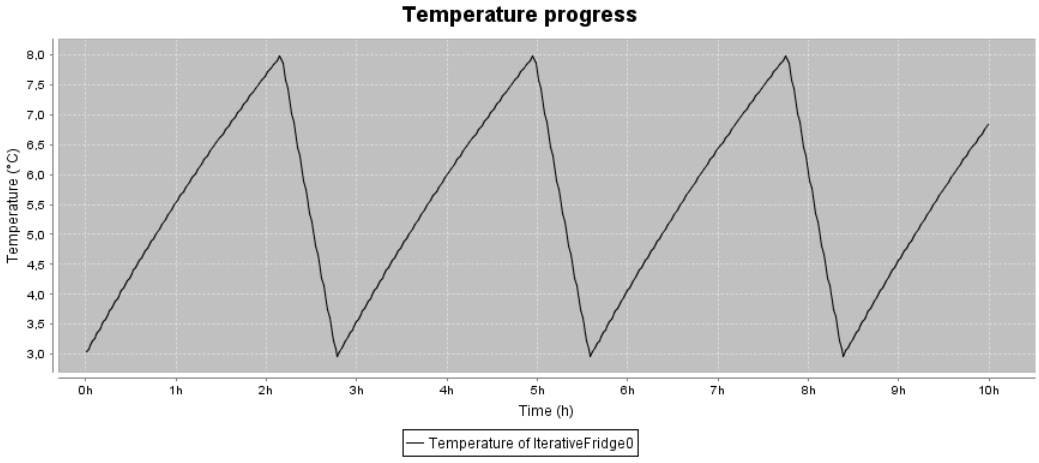
$$\tau_{warming}(Q) = -ln\left(\frac{T_{max} - T^O + \eta\frac{Q}{A}}{T_{min} - T^O + \eta\frac{Q}{A}}\right) \cdot \frac{m_c}{A} \cdot 60\,. \tag{3.4}$$

In [34] wurde für die Berechnung von $\tau_{warming}$ grundsätzlich $Q = q_{warming} = 0\,W$ angenommen. Hier ist Q jedoch trotzdem als Parameter in der Gleichung enthalten, um später einfacher auf andere Gerätetypen wie Heizungen usw. eingehen zu können, da diese dann unter Umständen (auch) in der Wärmphase Energie verbrauchen. Da τ in der Modellgleichung in Minuten gemessen wird, berechnen die obigen Formeln ebenfalls die Zeitspannen in Minuten. Die Längen der Phasen *Kühlen* und *Aufwärmen* betragen demnach mit den Standardparametern nach Berechnung mittels obiger Formeln

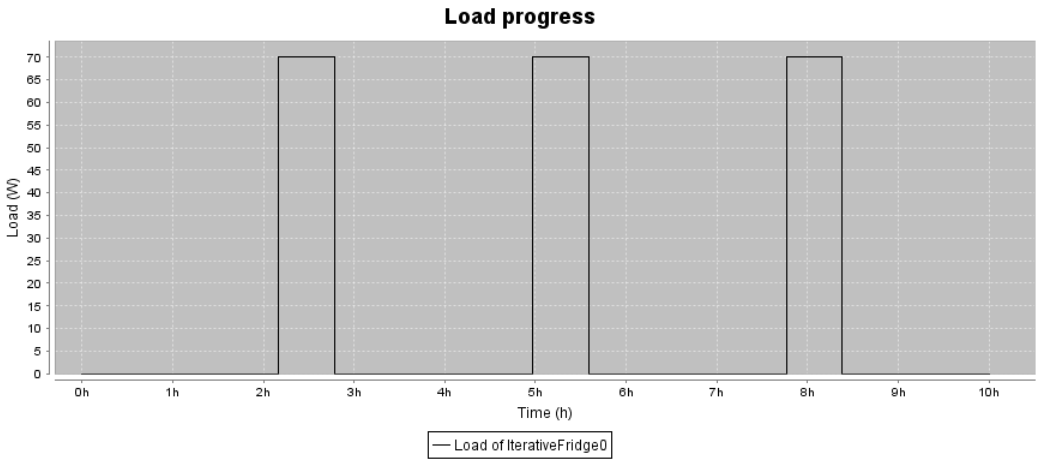
$$\begin{aligned} \tau_{cooling} &\approx 36.645 min\,, \\ \tau_{warming} &\approx 129.883 min\,. \end{aligned}$$

Für die gleichen Parameter ergeben sich nach einer (zeitdiskreten) Simulation des

Modells in SimKit die in Abbildung 3.3 dargestellten Verlaufskurven für die Innentemperatur sowie aufgenommene Leistung eines Kühlgerätes. Nach Messung der



(a)



(b)

Abbildung 3.3: Standardverlauf der Temperatur- (a) und Lastkurve (b) eines modellierten Kühlgerätes.

Zeitpunkte der Phasenwechsel im Verlauf der Simulation ergibt sich

$$\begin{aligned} \tau_{cooling} &= 37min\,, \\ \tau_{warming} &= 130min\,. \end{aligned}$$

Die leichten Unterschiede zu den oben berechneten Werten sind in der zeitlichen Auflösung der zeitdiskreten Simulation (hier: 1 Minute) und der daraus resultieren-

den Rundung begründet.
An dieser Stelle soll die Darstellung der Kurvenverläufe allein dem Verständnis des Verhaltens des Modells dienen, die Implementierung der diesen Ergebnissen zugrunde liegenden Simulation wird in den folgenden Abschnitten beschrieben.

3.3.1 Ereignisgraph des iterativen Kühlschrankmodells

Das im vorhergehenden Abschnitt beschriebene Verhalten eines Kühlgerätes lässt sich mit Hilfe der vorgestellten Formel 3.1 leicht in einen Ereignisgraphen übersetzen. Abbildung 3.4 zeigt den entstandenen Graphen.

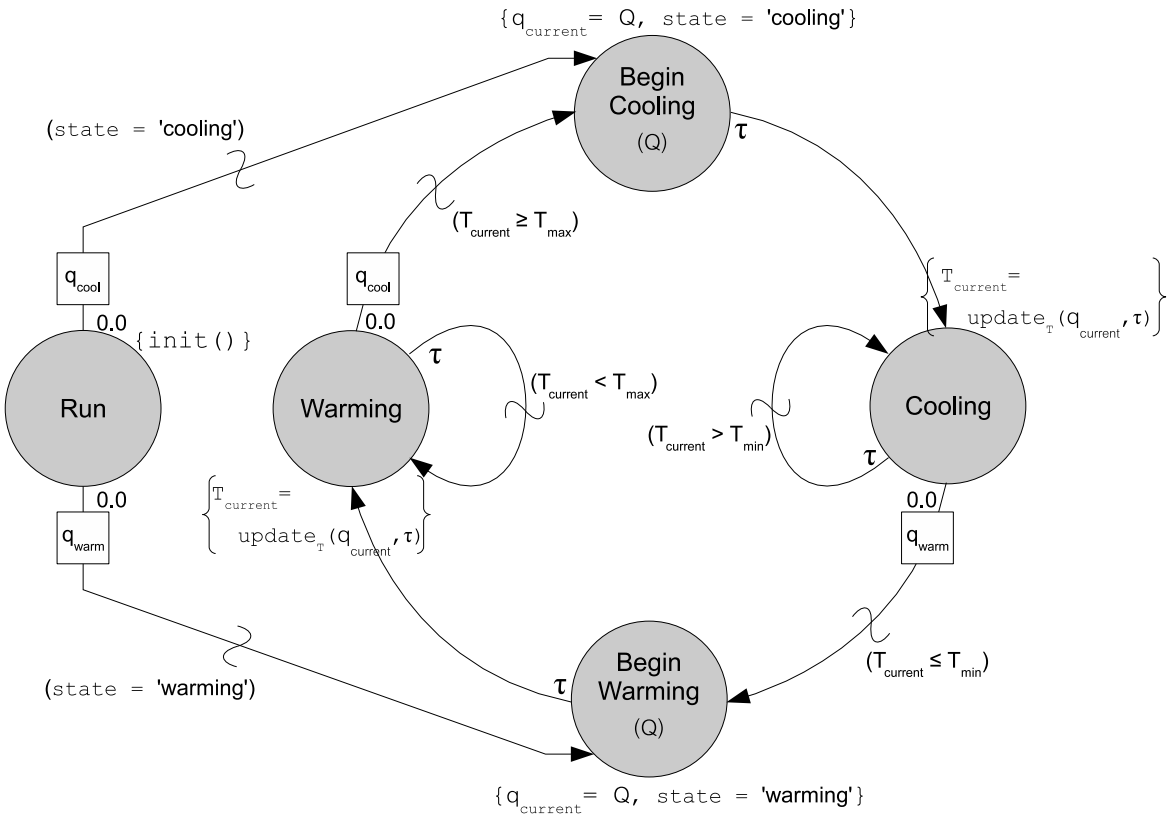


Abbildung 3.4: Ereignisgraph des iterativen Kühlschrankmodells.

Zu Beginn der Simulation tritt automatisch das Ereignis Run auf, welches sozusagen

als Starter des Modells dient und weitere Ereignisse einplant. In diesem Fall initialisiert Run zunächst die Parameter das Modells (insbesondere wird die Startphase mittels einer definierbaren Belegung von $state \in \{cooling, warming\}$ festgelegt) und schaltet dann abhängig vom gesetzten Startzustand unmittelbar in den Kühl- oder Aufwärmbetrieb. Den entsprechenden Ereignissen BeginCooling und BeginWarming wird die passende Last $q_{cooling}$ bzw. $q_{warming}$ übergeben, nach derer der weitere Temperaturverlauf berechnet wird. Die Last hätte im Code der entsprechenden Klasse auch intern als Zustandsvariable gesetzt und wieder ausgelesen werden können, das Übergeben als Parameter dient jedoch als Vorbereitung für spätere Überlegungen, in denen eventuell Zwischenstufen $Q \neq q_{cooling} \neq q_{warming}$ zur externen, feingranularen Steuerung des Kühlgerätes verwendet werden. Die Ereignisse BeginCooling und BeginWarming speichern also die empfangene Last, aktualisieren die Zustandsvariable *state* und planen dann das Folgeereignis Cooling bzw. Warming nach τ Zeiteinheiten ein. Dies ist die bereits im vorhergehenden Abschnitt angesprochene Stelle des Modells, an der die eigentlich ereignisdiskrete Simulation auf zeitdiskreter Basis ausgeführt wird. Dazu kann der Verzögerungsparameter τ hier als Simulationstakt identifiziert werden, da er direkt der Zeitspanne $\tau = [t_{i-1}, t_i]$ aus Formel 3.1 entspricht. Die Zustände Cooling und Warming berechnen bei Eintritt nun nach dieser Formel die seit dem letzten Zeitpunkt veränderte Temperatur und planen abhängig davon entweder sich selbst erneut nach dem gleichen Intervall τ ein, oder schalten in den entgegengesetzten Betrieb durch Einplanen des komplementären Ereignisses BeginWarming oder BeginCooling.

Die Entscheidung, die Phasen *Kühlen* und *Aufwärmen* durch jeweils ein Einstiegs- und ein weiterführendes Ereignis zu modellieren, liegt in der internen statistischen Verarbeitung der Modellvariablen im Simulationsframework SimKit begründet: Bei jeder Änderung von beobachteten Variablen wird ein internes Ereignis ausgelöst, welches die Aufzeichnung der Wertänderung samt eines Zeitstempels in den überwachenden Listenern verursacht. Da die Ereignisse Cooling und Warming jedoch bis zu einem Phasenwechsel für jeden Simulationstakt wieder eingeplant werden, hätte das Setzen der Last in diesem Ereignis zur Folge, dass permanent Änderungen dieser Variablen aufgezeichnet würden, obwohl sich ihr Wert eigentlich nur einmal, und zwar zu Beginn der Phase geändert hat. Daher wurde die Aktualisierung der Last in ein eigenes Ereignis BeginCooling bzw. BeginWarming ausgelagert.

Die in Abschnitt 3.3 dargestellte Abbildung 3.3 der Verlaufskurven wurde durch Implementierung und Ausführung dieses Ereignisgraphen in SimKit in Verbindung mit der bereits erwähnten Bibliothek *JFreeChart* generiert.

3.4 Abschnittsweise Linearisierung des Modells

Da sich das Verhalten des im vorhergehenden Abschnitt beschriebenen iterativen Modells nur zu den Phasenwechseln wesentlich ändert und dazwischen grob gesehen linear verläuft, lässt sich die iterative Modellgleichung wie folgt vereinfachen.
Nimmt man einen exakt linearen Temperaturverlauf zwischen den Phasenwechseln sowie eine konstante Last $q_{cooling}$ bzw. $q_{warming}$ je Phase und die sich daraus ergebenen konstanten Zeitspannen $\tau_{cooling}\,(q_{cooling})$ und $\tau_{warming}\,(q_{warming})$ für die beiden Phasen an, so lässt sich der Temperaturverlauf in einer Phase wie folgt berechnen (leicht verändert entnommen aus [34]):

$$T_{current}(T_{prev}, \tau_{elap}) = T_{prev} + a \cdot \tau_{elap}$$

$$\text{mit} \quad a = \begin{cases} a_c = \frac{T_{min} - T_{max}}{\tau_{cooling}} < 0 & \text{für} \quad state = cooling\,, \\ a_w = \frac{T_{max} - T_{min}}{\tau_{warming}} > 0 & \text{für} \quad state = warming\,. \end{cases} \tag{3.5}$$

In dieser linearen Funktion bezeichnet T_{prev} eine zuvor berechnete Temperatur in der gleichen Phase und der Parameter τ_{elap} die Länge des Intervalls $[t_{prev}, t_{current}]$, also genau die Zeitspanne seit der Berechnung von T_{prev}. Die Werte a_c und a_w beschreiben die Steigung des linearen Temperaturverlaufes in der jeweiligen Phase, welche durch die Fallunterscheidung kenntlich gemacht wird. Es ist zu beachten, dass in der Funktion davon ausgegangen wird, dass sich die Zeitpunkte t_{prev} und $t_{current}$ in der gleichen Phase befinden. Weiterhin ist bei der Verwendung von Zwischenstufen der Leistungsaufnahme (also etwa $q_{warming} < q_{current} < q_{cooling}$) zu beachten, dass die aktuelle Leistungsaufnahme $q_{current}$ die gleiche sein muss, die für die Berechnung des in der Gleichung verwendeten $\tau_{cooling}$ bzw. $\tau_{warming}$ verwendet wurde, da diese Zeiten sonst unstimmig sind und die Funktion falsche Ergebnisse liefert.
Weiterhin auf Basis des linearen Temperaturverlaufes herzuleiten ist die im Gegensatz zu Formel 3.2 vereinfachte Berechnungsvorschrift für die benötigte Zeitspanne $\tau_{req-lin}$, um ausgehend von einer Temperatur T_{from} eine Zieltemperatur T_{dest} zu erreichen. Sie ist durch folgende Gleichung gegeben:

$$\tau_{req-lin}(T_{from}, T_{dest}) = \begin{cases} \frac{T_{dest} - T_{from}}{a_c} & \text{für} \quad T_{dest} < T_{from} \\ \frac{T_{dest} - T_{from}}{a_w} & \text{sonst.} \end{cases} \tag{3.6}$$

Diese Gleichung geht wie die vorhergehende ebenfalls von feststehenden Leistungs-

aufnahmen $q_{cooling}$ und $q_{warming}$ aus. Bei der Verwendung von Zwischenstufen müssten hier ebenfalls $\tau_{cooling}$ bzw. $\tau_{warming}$ auf Basis der gewünschten Leistungsaufnahme neu ermittelt, oder stattdessen die nichtlineare Form dieser Gleichung, also Formel 3.2, verwendet werden.

Laut Stadler et al. ist die Annahme des linearen Temperaturverlaufes während einer Phase im Vergleich zum iterativen Modell akzeptabel, da im Vergleich der beiden Berechnungen für den Standardfehler SE gilt:

$$SE \quad < \quad 0.00023\,.$$

3.4.1 Ereignisgraph des linearisierten Modells

Wie beim iterativen Modell, so lässt sich auch die lineare Betrachtungsweise in einen Ereignisgraphen übersetzen, Abbildung 3.5 zeigt das Ergebnis.

Analog zu Abbildung 3.4 startet die Simulation des Modells mit dem Ereignis Run und schaltet von dort aus basierend auf den Initialparametern in die Phase *Kühlen* oder *Aufwärmen*. Im Vergleich zum Graphen des iterativen Modells existiert jedoch nur noch jeweils ein Zustand für die beiden Phasen. Dies liegt daran, dass nicht mehr in jedem Simulationstakt ein Ereignis ausgelöst werden muss, sondern nur zu den Phasenwechseln, weshalb die Zuweisung der Last simultan zur Aktualisierung der Temperatur geschehen kann. Zu beachten ist hier allerdings die Reihenfolge der Operationen: Zunächst wird die aktuelle Temperatur $T_{current}$ basierend auf der bisherigen Temperatur, der Zeitspanne seit dem letzten Aufruf sowie der aktuellen Leistungsaufnahme über die Formel 3.5 berechnet. Anschließend wird die Leistungsaufnahme $q_{current}$ für das folgende Zeitintervall auf den übergebenen Parameter Q gesetzt. Dann wird mittels der Formel 3.6 die Zeitspanne ermittelt, die das Modell in dieser Phase bleiben muss, um die gewünschte Zieltemperatur T_{min} bzw. T_{max} (je nach Phase) zu erreichen. Dieser Wert wird nun als Verzögerung zur Einplanung des komplementären Ereignisses verwendet.

Mit den bereits in Tabelle 3.1 vorgestellten Standardparametern ergibt sich für das lineare Modell der in Abbildung 3.6 dargestellte Verlauf für Temperatur und Leistungsaufnahme.

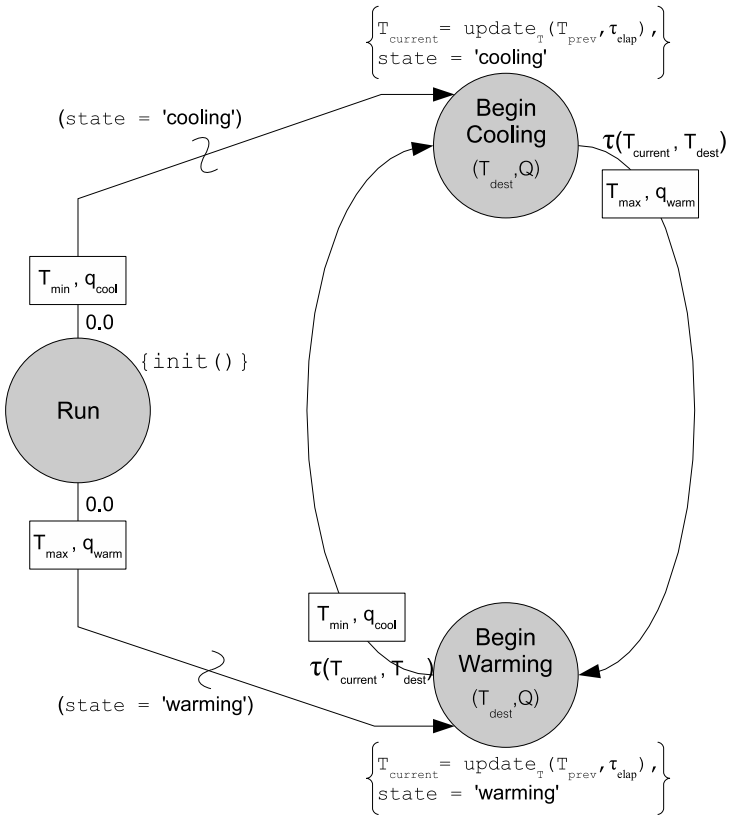
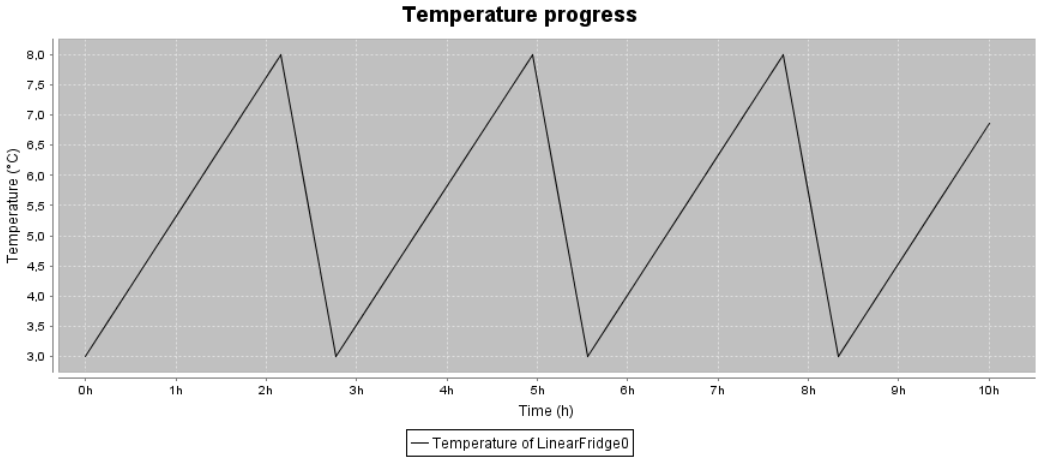
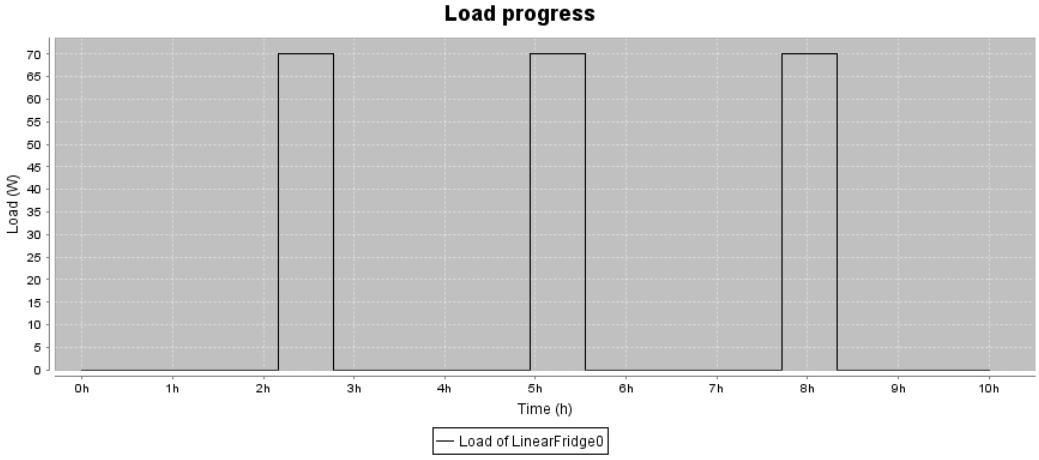


Abbildung 3.5: Ereignisgraph des linearisierten Kühlschrankmodells.



(a)



(b)

Abbildung 3.6: Standardverlauf der Temperatur- (a) und Lastkurve (b) des linearen Modells.

3.4.2 Kompakter Ereignisgraph des linearisierten Modells

Der im vorhergehenden Abschnitt beschriebene Graph lässt sich allerdings noch weiter vereinfachen. Bei genauer Betrachtung fällt auf, dass die Ereignisse BeginCooling und BeginWarming sich allein durch die ihnen übergebenen Parameter unterscheiden. Daher ist es möglich, diese Ereignisse zu einem einzelnen zusammen zu fassen, Abbildung 3.7 zeigt den entstehenden Ereignisgraphen.

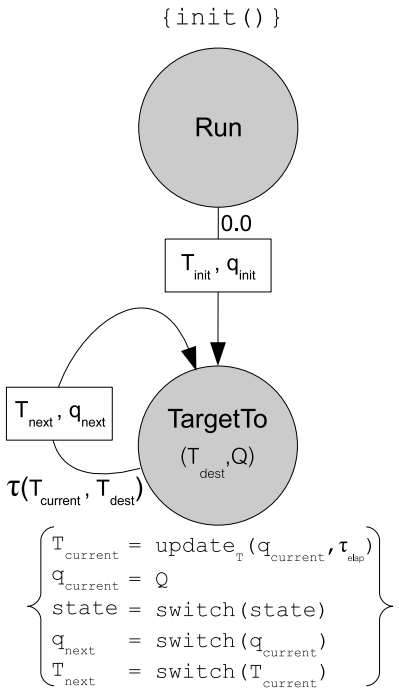


Abbildung 3.7: Kompakter Ereignisgraph des linearisierten Modells.

Die Simulation startet wie üblich in dem Ereignis Run, jedoch wird hier keine Fallunterscheidung des nachfolgend einzuplanenden Ereignisses vorgenommen. Stattdessen wird eine gewünschte Zieltemperatur auf Basis der Initialparameter berechnet, und diese Zieltemperatur zusammen mit einer entsprechenden Leistungsaufnahme an das einzuplanende Ereignis TargetTo übergeben. Dieses Ereignis arbeitet nun ähnlich wie die Ereignisse BeginCooling und BeginWarming des vorhergehenden Abschnittes, nur dass es statt der Einplanung seines Komplementes erneut sich selbst mit abweichenden Parametern einplant. Die in diesem Ereignis durchzuführenden

Operationen verdeutlichen dies: zunächst wird wie gehabt die aktuelle Temperatur auf Basis der Leistungsaufnahme der nun abgeschlossenen Phase, welche in $q_{current}$ gespeichert ist, und der vergangenen Zeit τ_{elap} seit dem letzten Ereignis berechnet. Anschließend wird die aktuelle Leistungsaufnahme $q_{current}$ aktualisiert, indem sie auf den übergebenen Wert Q für die nun folgende Phase gesetzt wird. Auch die Zustandsvariable *state* wird in Bezug auf die nun folgende Phase aktualisiert. Zuletzt werden noch die Parameter q_{next} und T_{next} sowie die Verzögerung τ für die nächste Einplanung des Ereignisses ermittelt. Die drei switch() Anweisungen geben jeweils das Komplement des ihnen übergebenen Parameters zurück (beispielsweise gilt also $\mathsf{switch}(T_{min}) = T_{max}$). Der Verzögerungsparameter τ wird nach Formel 3.6 berechnet und bezeichnet genau die Zeit, die das System in dem aktuellen Zustand *state* bleiben muss, bis die gewünschte Temperatur T_{dest} erreicht ist.

Der Verlauf für Temperatur und Leistungsaufnahme nach einer Simulation mit diesem Ereignisgraphen entsprechen denen des linearisierten Graphen im vorherigen Abschnitt. Die Zeiten $\tau_{cooling}$ und $\tau_{warming}$ sind bei allen drei Modellbetrachtungen identisch, da überall die gleiche Berechnungsvorschrift verwendet wird.
Für die Durchführung der Simulation und weitergehende analytische Berechnungen ist die lineare Betrachtungsweise generell sehr hilfreich, da sie eine starke Vereinfachung der Berechnungen ermöglicht. Die sich ergebende Simulation kann nun tatsächlich als ereignisorientiert betrachtet weren, da kein eindeutiger Simulationstakt mehr identifiziert werden kann. Der Zustand des Systems ändert sich nur noch zu den Phasenwechseln. Dies bedingt allerdings auch, dass eine Darstellung des mittleren Temperaturverlaufes mehrerer simulierter Kühlgeräte nicht mehr so leicht zu realisieren ist, da die Berechnung der Mittelwerte die Kenntnis der Temperaturen aller Entitäten zu beliebigen Zeitpunkten erfordert. Da diese Werte aber nicht wie im iterativen Modell in jedem Zeitschritt aktualisiert werden, müssen sie manuell berechnet werden. Das gilt insbesondere bei einer Population mit gestreuten Parametern und demzufolge unterschiedlichen Zeitpunkten für die jeweiligen Phasenwechsel. Wird also eine Visualisierung des Temperaturverlaufes gewünscht, so erhöht sich der Berechnungsaufwand. Allerdings ist die Berechnung der Temperatur in Formel 3.5 eine einfache lineare Gleichung, wodurch sich die Ausführungszeit der Simulation nicht wesentlich erhöht (die Größenordnung liegt bei $10\% - 20\%$).

3.5 Implementierung der Modelle

Die Verlaufskurven in den vorhergehenden Abschnitten wurden mittels Simulation der beschriebenen Modelle durch das Framework SimKit erzeugt. In diesem Abschnitt soll gezeigt werden, wie die korrespondierenden Java Klassen in SimKit angelegt wurden und wie sie miteinander interagieren. Abbildung 3.8 zeigt die Klassen des Basismodells.

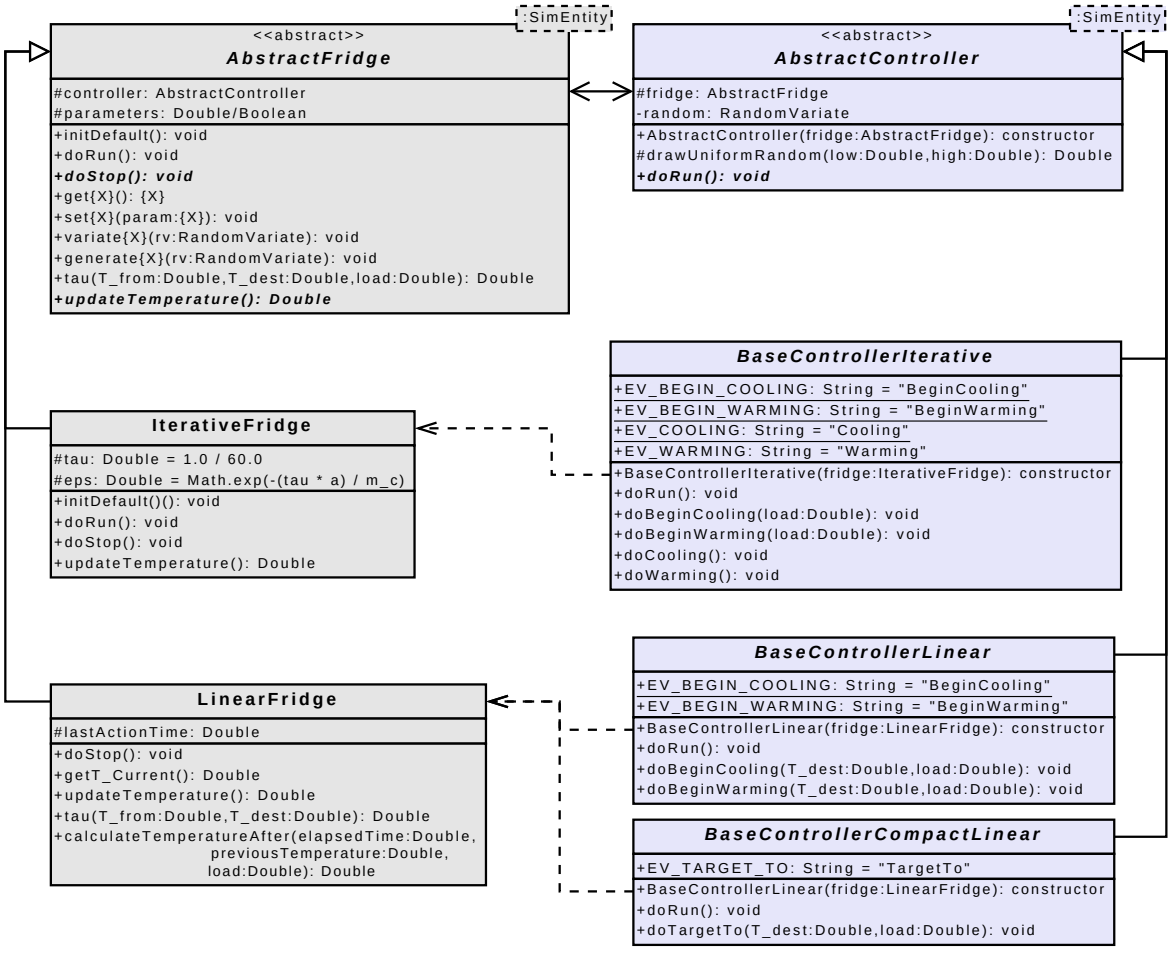


Abbildung 3.8: Klassendiagramm des Modells (Basis).

Hinsichtlich der zu entwerfenden bzw. implementierenden Steuerungsstrategien wurde zwischen der Entität Kühlgerät (*Fridge*, linke Seite der Grafik) und seinem *Controller* (rechte Seite) unterschieden. Zentrale Basisklasse in der Hierarchie der Kühl-

geräte ist die abstrakte Klasse AbstractFridge, die von der Klasse `SimEntityBase` aus SimKit abgeleitet ist und somit eine simulierbare Entität repräsentiert. Von ihr erben die Spezialisierungen IterativeFridge und LinearFridge. Daneben gibt es eine Hierarchie von *Controllern* (ebenfalls von SimEntityBase abgeleitet), zentrale Klasse ist hier AbstractController, von welcher die Klassen BaseControllerIterative, BaseControllerLinear und BaseControllerCompactLinear erben. Die *Fridge* Klassen implementieren den formalen Teil des Modells, während die *Controller* Klassen die Ereignisgraphen realisieren. Das Modell ist nur in Kombination beider Teile lauffähig, daher gibt es eine Abhängigkeitsbeziehung zwischen den Entitäten *Fridge* und *Controller*, wobei diese Abhängigkeit je nach Spezialisierungsebene angepasst wird (so benötigt ein BaseControllerIterative z. B. ein IterativeFridge, und nicht etwa eine Klasse aus dem linearen Teil des Hierarchiebaums). Im Folgenden werden die einzelnen Klassen und ihre Aufgaben näher beschrieben.

3.5.1 Die Klasse `AbstractFridge`

Wie bereits angedeutet beschreibt die Klasse AbstractFridge den formalen Teil des mathematischen Modells, der sowohl für den iterativen, als auch den linearen Zweig gilt. Dazu zählen zum einen die einzelnen Geräteparameter (wie m_c, A etc.), aber auch einige grundlegende ausformulierte sowie abstrakte Methoden, die von allen erbenden Klassen verwendet werden bzw. implementiert werden müssen.

Parameter:

AbstractController controller
Dieses Feld stellt einen Teil der Abhängigkeitsbeziehung zu den *Controller* Entitäten dar. Außerdem ist es über dieses Feld möglich, von einem AbstractFridge Objekt auf die Instanz des ihm zugewiesenen *Controllers* zuzugreifen.

Double/Boolean parameters
Diese Bezeichnung steht repräsentativ für die enthaltenen Geräteparameter wie m_c, A etc.

Methoden:

public void initDefault()
Setzt die Geräteparameter auf die voreingestellten Standardwerte, welche über hier nicht dargestellte statische Klassenvariablen definiert werden können.

public void doRun()
Repräsentiert das Run Ereignis, wird zum Start der Simulation ausgeführt. Hier wird zum einen der initiale Wert von $T_{current}$ bekannt gegeben, sodass observierende Listener den Startwert ihren Statistiken zuweisen können. Zum anderen wird an dieser Stelle ein PropertyChangeListener angelegt, der die Ereignisliste observiert und bei Auftreten des *Stop* Ereignisses (entspricht dem Beenden der Simulation) benachrichtigt wird, um dann die Methode doStop() aufzurufen.

public abstract void doStop()
Diese Methode muss von erbenden Klassen implementiert werden. Sie wird zum Ende der Simulation aufgerufen und soll der Entität eine Möglichkeit geben, den Stand ihrer Parameter zum Zeitpunkt des Aufrufs ein letztes Mal zu aktualisieren und den observierenden Listenern mitzuteilen, damit diese die Abschlusswerte ihren Statistiken zuweisen können. Dies ist insbesondere beim linearen Modell notwendig, da dort die Parameterbelegungen nicht in jedem Zeitschritt aufgezeichnet werden und ein zwischenliegendes Ende der Simulation ansonsten zu verfälschten Statistiken führen würde.

public $\{X\}$ get$\{X\}$()
Steht repräsentativ für alle Getter-Methoden dieser Klasse.

public void set$\{X\}$($\{X\}$ param)
Steht repräsentativ für alle Setter-Methoden dieser Klasse.

public void variate$\{X\}$(RandomVariate rv)
Diese Methode multipliziert den Parameter $\{X\}$ mit einer Zufallszahl, die aus dem übergebenen Generator gewonnen wird.

public void generate$\{X\}$(RandomVariate rv)
Diese Methode setzt den Parameter $\{X\}$ auf die Zufallszahl, die aus dem übergebenen Generator gewonnen wird.

public void tau(Double T_{from}, Double T_{dest}, Double load)
Realisiert die weiter oben eingeführte Formel 3.2. Wird zudem zur Berechnung von $\tau_{cooling}$ und $\tau_{cooling}$ verwendet (Formel 3.3 und Formel 3.4).

3.5.2 Die Klasse `IterativeFridge`

`IterativeFridge` realisiert das iterative Modell und erweitert die Klasse `AbstractFridge` um folgende Parameter und Methoden:

Parameter:

Double tau
Dieses Feld beschreibt den Parameter τ für Formel 3.1 und wird standardmäßig mit $\tau = 1$ initialisiert, was einer Minute entspricht und somit den Takt der hier noch zeitdiskreten Simulation definiert.

Double eps
Dieses Feld stellt den Parameter ε aus Formel 3.1 dar und wird beim Starten der Simulation auf Basis des obigen Wertes τ berechnet.

Methoden:

public void initDefault()
Erweitert die gleichnamige Methode der Superklasse um die Initialisierung des Standardwertes für das Feld tau.

public void doRun()
Erweitert die gleichnamige Methode der Superklasse um die Berechnung des Feldes eps.

public abstract void doStop()
Das iterative Modell zeichnet ohnehin zu jedem Zeitschritt der Simulation die Parameteränderungen auf, daher ist an dieser Stelle keine weitere Operation notwendig und die Methode ist leer.

public Double updateTemperature()
Implementiert die gleichnamige abstrakte Methode der Superklasse und realisiert Formel 3.1. In dem iterativen Modell wird diese Methode also in jedem Simulationsschritt aufgerufen, um die aktuelle Temperatur zu berechnen und festzulegen. Dies hat zusätzlich zur Folge, dass die Statistiken, die diese Entität observieren, benachrichtigt werden und eine Mitteilung über die Änderung erhalten.

3.5.3 Die Klasse `LinearFridge`

`LinearFridge` realisiert das linearisierte Modell und erweitert die Klasse `AbstractFridge` um folgende Parameter und Methoden:

Parameter:

Double lastActionTime
Dieses Feld wird nach jeder Berechnung aktualisiert und repräsentiert den Parameter τ_{elap} aus Formel 3.5.

Methoden:

public abstract void doStop()
Ruft ein letztes Mal vor Ende der Simulation die Methode updateTemperature() auf.

public Double getT_current()
Überschreibt die gleichnamige Methode der Superklasse um eine genaue Berechnung des Wertes $T_{current}$ mittels der Methode calculateTemperatureAfter(Double tau, Double T_current, Double load), da dieser Wert hier gemäß der linearen Betrachtungsweise und des sich daraus ergebenden ereignisorientierten Verhaltens nur zu den Phasenwechseln aktualisiert wird und eine normale get Methode inmitten einer Phase einen veralteten Wert liefern würde. Die Überschreibung dieser Methode realisiert die zum Schluss von Abschnitt 3.4.2 angesprochene manuelle Berechnung des Temperaturwertes zu beliebigen Zeitpunkten zur Darstellung des mittleren Temperaturverlaufes einer Population von simulierten linearen Kühlgeräten.

public Double updateTemperature()
Implementiert die gleichnamige abstrakte Methode der Superklasse und aktualisiert zunächst $T_{current}$ mittels der Methode calculateTemperatureAfter(Double tau, Double T_current, Double load) und setzt anschließend lastActionTime auf die aktuelle Simulationszeit.

calculateTemperatureAfter(Double tau, Double T_current, Double load)
Realisiert die lineare Formel 3.5.

public void tau(Double T_{from}, Double T_{dest})
Realisiert die lineare Formel 3.6.

3.5.4 Die Klasse `AbstractController`

Die Klasse AbstractController stellt die abstrakte Basis der *Controller* Entitäten dar und enthält die folgenden generellen Felder und Methoden.

Parameter:

AbstractFridge fridge
Dieses Feld stellt den zweiten Teil der Abhängigkeitsbeziehung zwischen *Fridge* und *Controller* Entitäten dar. Über dieses Feld ist es zudem möglich, von einem AbstractController Objekt auf die Instanz des ihm zugewiesenen *Fridge* zuzugreifen.

static RandomVariate random
Dies ist ein Zufallszahlengenerator, der vorbereitend für spätere Steuerungsstrategien enthalten ist und gleichverteilte Zufallszahlen erzeugt.

Double lastLow, Double lastHigh
Aktuelle Parameter des Zufallszahlengenerators. Sie definieren das Intervall, in dem Werte generiert werden.

Methoden:

public Double drawUniformRandom(low, high)
Konfiguriert zunächst den internen Zufallszahlengenerator auf das gewünschte Intervall, falls notwendig, und liefert dann den nächsten Zufallswert des Generators zurück.

public abstract void doRun()
Repräsentiert das Run Ereignis, wird zum Start der Simulation ausgeführt. Diese Methode ist abstrakt, muss also von erbenden Klassen mit Inhalt gefüllt werden.

3.5.5 Implementierung der konkreten Controller

Die Klassen BaseControllerIterative, BaseControllerLinear und BaseControllerCompactLinear erweitern AbstractController um die Realisierung der ihnen zugeordneten Ereignisgraphen, indem sie für jedes Ereignis der Ereignisgraphen je eine für SimKit spezifische Identifikations-Zeichenkette sowie eine Methode besitzen, in welcher die Operationen des Ereignisses auf der assoziierten *Fridge* Entität ausgeführt werden. Die Implementierung ist trivial und wird daher hier nicht näher beschrieben.

Kapitel 4

Steuerungssignale

Im vorhergehenden Kapitel wurde das Basismodell der Kühlgerätesimulation vorgestellt. Um nun ein extern gesteuertes Lastmanagement zu erreichen, müssen die Controller Möglichkeiten zur Modifikation ihres Standardverhaltens bieten. Es werden nun zunächst zwei Signale von Stadler et al. samt Implementation beschrieben, danach folgen Überlegungen zu Strategien zur Dämpfung der durch die Steuerung entstehende Synchronisierung der Geräte.

4.1 Steuerung nach „Direct Storage Control“

Die einfachste denkbare Steuerung von Kühlgeräten ist die direkte Beeinflussung der Aktivität des Gerätes in Bezug auf die Phasen *Kühlen* bzw. *Aufwärmen*. In [34, Abs.5(1)] wurde diese Überlegung unter der Überschrift „Direct Storage Control“ (DSC) umgesetzt. Dieses Szenario sieht lediglich zwei komplementäre Steuerungssignale load thermal storage und unload thermal storage sowie einen Streuungsfaktor spread vor. Erhält ein Kühlgerät (bzw. sein Controller) das Signal load thermal storage, so unterbricht es einen eventuell stattfindenden Aufwärmvorgang, beginnt die Kühlung bis zur Minimaltemperatur und füllt somit seinen thermischen Speicher. Umgekehrt veranlasst das Signal unload thermal storage ein Gerät dazu, eine eventuell laufende Kühlung zu unterbrechen und sich auf seine Maximaltemperatur aufzuwärmen. Der Parameter spread beschreibt dabei den maximalen Zeitraum, den das Gerät nach Erhalt des Steuerungssignales abwarten darf, bis es die geforderte Operation ausführt. Wird diese Verzögerung etwa durch einen in den Controllern enthaltenen Zufallszahlengenerator auf einen zufälligen Zeitpunkt im Intervall $[t_{notify}, t_{notify} + spread]$ gesetzt (wobei t_{notify} den Zeitpunkt des Erhalts des Signals

beschreibt), so wird damit bei mehreren zu steuernden Geräten eine Reaktionsverteilung über das beschriebene Intervall erreicht. Dies bewirkt eine Glättung in der gemittelten Lastkurve der Geräte.

4.1.1 Implementierung DSC

Um die Steuerung DSC in der bereits implementierten Basisversion des Modells zu realisieren, wurde zunächst ein Interface Idsc angelegt, welches die Steuerungssignale repräsentiert. Listing 4.1 zeigt den Inhalt des Interfaces.

Listing 4.1: Interface Idsc

```
1 public interface Idsc {
2   public final static String EV_LOAD_THERMAL_STORAGE =
3           "LoadThermalStorage";
4   public final static String EV_UNLOAD_THERMAL_STORAGE =
5           "UnloadThermalStorage";
6
7   public abstract void doLoadThermalStorage(Double spread);
8   public abstract void doUnloadThermalStorage(Double spread);
9 }
```

Die Schnittstelle definiert die beiden Steuerungssignale als Methoden im Stil der SimEntity Ereignismethoden und beinhaltet die dazugehörigen Identifikations-Zeichenketten. Die Methoden erhalten jeweils den Wert spread als Parameter. Dieses Interface kann nun verwendet werden, um die bestehenden Controller der verschiedenen Modelltypen um die Steuerung DSC zu erweitern.

Abbildung 4.1 zeigt den Ereignisgraphen der erweiterten Version des iterativen Controllers. Die bereits bekannten Ereignisse und Operationen der Basisversion sind ausgegraut abgebildet, um die Änderungen hervorzuheben. Zudem wurde das Run Ereignis sowie die Angabe der Operationen der Ereignisse Cooling und Warming aus Gründen der Übersichtlichkeit entfernt.

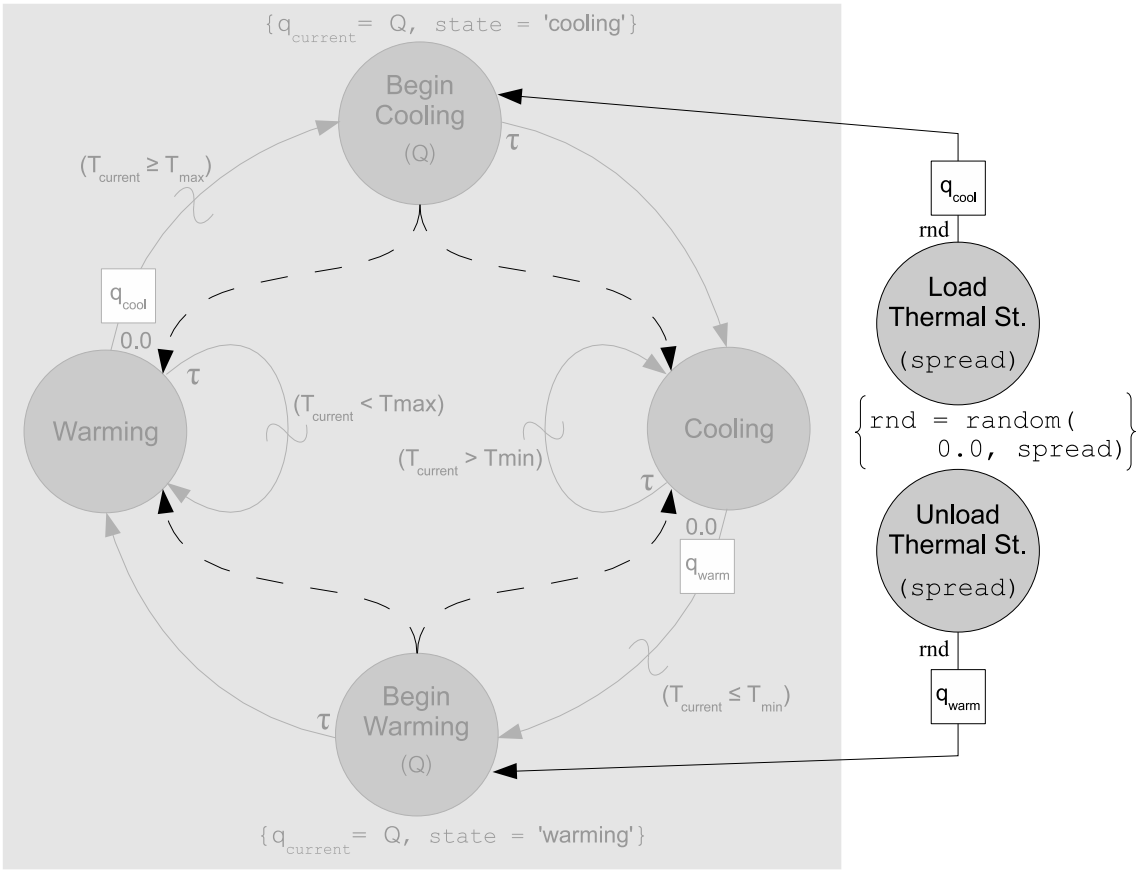


Abbildung 4.1: Ereignisgraph des DSC-Controllers (iterativ).

Hinzugekommen sind die beiden Ereignisse des beschriebenen Interfaces. Sie planen ihrerseits die Ereignisse BeginCooling bzw. BeginWarming mit entsprechenden Parametern ein, um die DSC-Steuerung zu realisieren. Diese beiden Ereignisse haben zusätzlich löschende Kanten erhalten, die bei Eintritt des jeweiligen Ereignisses eventuell noch andere eingeplante Ereignisse dieses Controllers aus der Ereignisliste entfernen, um so den gewünschten Betrieb zu gewährleisten. Die Verzögerung *rnd* wird als gleichverteilte Zufallszahl aus dem Intervall $[0.0, spread]$ errechnet.

Abbildung 4.2 zeigt analog dazu die Erweiterungen des linearen Controllers. Die bekannten Teile befinden sich wiederum im ausgegrauten Bereich, hier sind allerdings nur die Ereignisse BeginCooling und BeginWarming als Anknüpfungspunkte

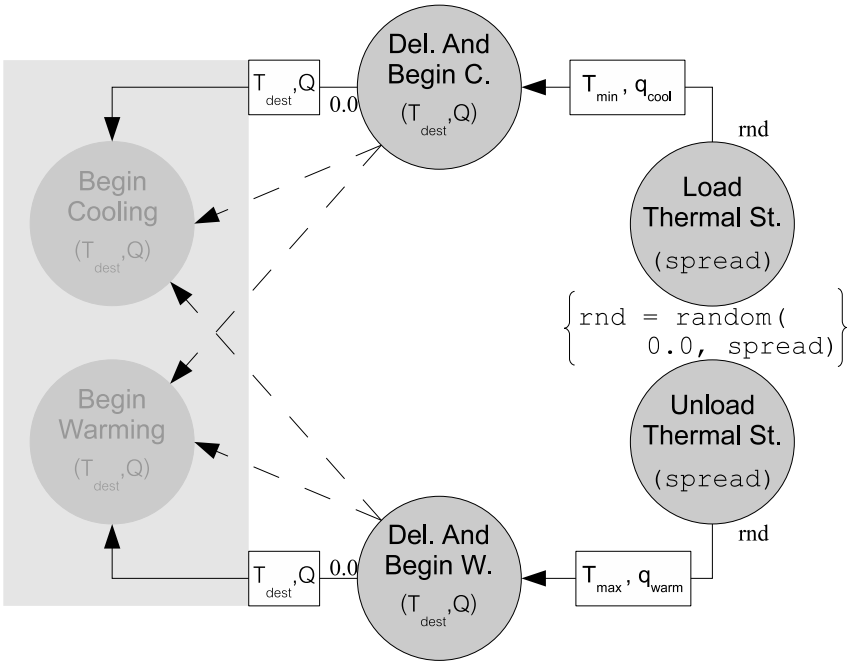


Abbildung 4.2: Ereignisgraph des DSC-Controllers (linear).

abgebildet. Wie bei dem iterativen Controller sind die beiden Ereignisse des Interfaces hinzugekommen. Die löschenden Kanten gehen jedoch von zwei separaten Ereignissen aus, die direkt vor den ursprünglich einzuplanenden Ereignissen stehen. Dieser Umweg musste gegangen werden, da das Einfügen der löschenden Kanten in die usprünglichen Ereignisse (wie bei der iterativen Version) negative Seiteneffekte hervorrufen würde, insbesondere im zeitnahen Umfeld eines Steuerungssignales. So würde beispielsweise ein durch ein Steuerungssignal mit der Verzögerung rnd eingeplantes Ereignis von dem Controller wieder gelöscht werden, wenn innerhalb des Intervalls $[t_{notify}, t_{notify} + rnd]$ (also nach Senden des Signals, aber vor der durch den spread Parameter verzögerten Aktion) ein regulärer Phasenwechsel statt fände. Dieser Effekt ist umso wahrscheinlicher, je größer der verwendete Wert spread ist. Umgekehrt konnte die Löschaktion aber auch nicht direkt in den Ereignissen des Interfaces statt finden, da löschende Kanten in SimKit nicht mit Verzögerung ausgeführt werden können und eine unmittelbare Löschung bei Eintritt des Kontrollsignales ebenfalls negative Seiteneffekte hätte. So würde beispielsweise ein im Intervall $[t_{notify}, t_{notify} + rnd]$ regulär eingeplanter Phasenwechsel durch die Löschung nicht

ausgeführt, was eine Verletzung das grundlegenden Constraints der Kühlgeräte zur Folge hätte, und zwar die Nichteinhaltung der Temperaturgrenzen $[T_{min}, T_{max}]$. Diese beiden problematischen Situationen konnten durch die eigens für den modifizierten Betrieb eingefügten Ereignisse umgangen werden, da so die Löschaktion nur durch ein Kontrollsignal ausgelöst wird, und zudem direkt vor Eintritt des modifiziert eingeplanten Phasenwechsels.

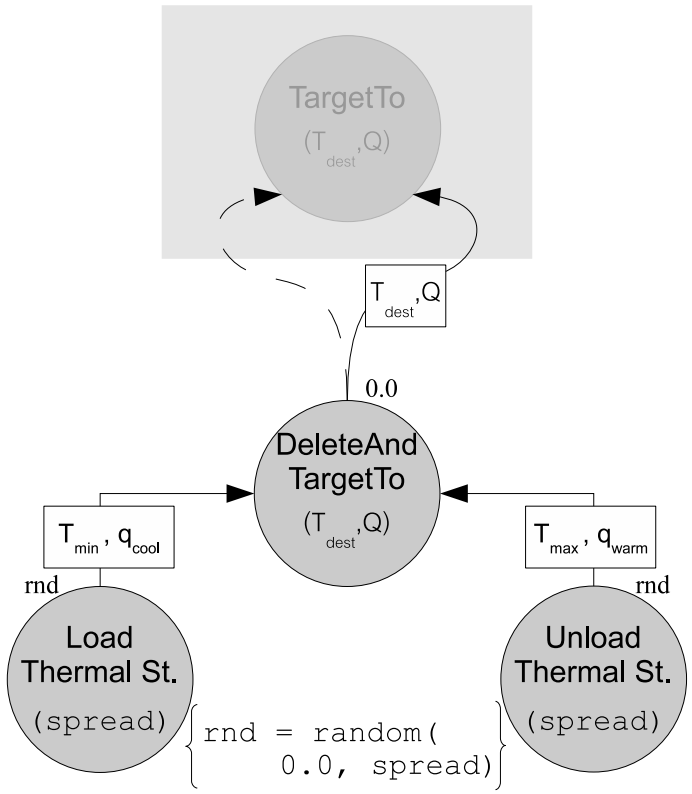


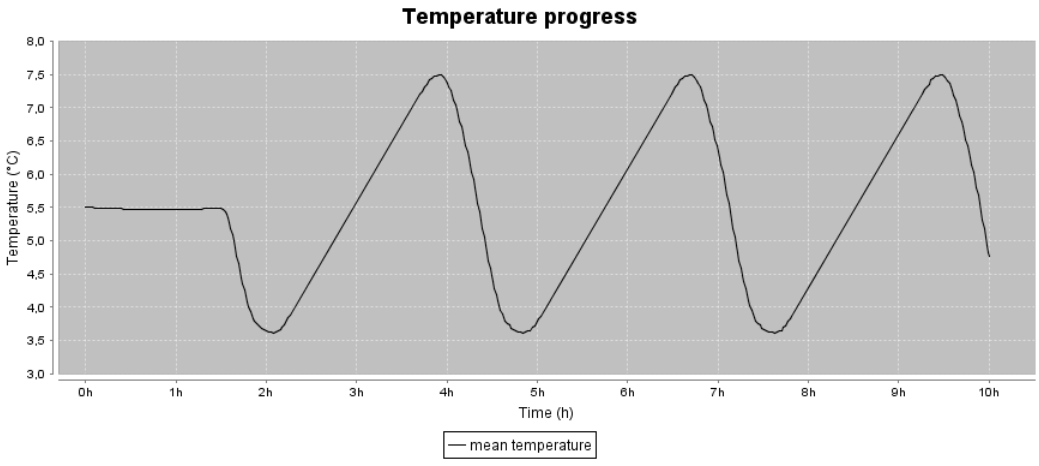
Abbildung 4.3: Kompakter Ereignisgraph des linearen DSC-Controllers.

Abbildung 4.3 beschreibt abschließend die erweiterte Version des kompakten Ereignisgraphen für den linearen Controller. Die Grafik wurde im Bereich der bekannten Elemente ebenfalls stark vereinfacht. Neu hinzugekommende Elemente sind die Ereignisse des Interfaces sowie eine löschende Kante mit der bereits beschriebenen Funktion. Auch hier wurde eine eigene Ebene zur Löschung angelegt, die Begrün-

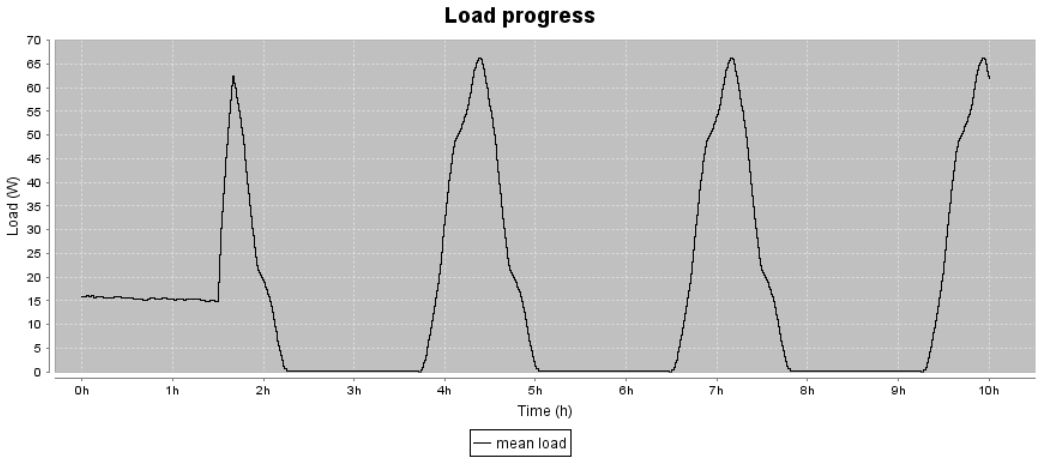
dung ist identisch zu der beim linearen Controller.
Die Beschreibung der konkreten Implementierung der Methoden und Parameter der erweiterten Controller wird an dieser Stelle aufgrund ihrer Trivialität vernachlässigt.

4.1.2 Simulationsergebnisse nach DSC-Steuerung

Abbildung 4.4 zeigt die Ergebnisse einer Simulation von 5000 Kühlgeräten mit dem linearen Controller (kompakte Version) und aktivierter DSC-Steuerung (Signal *load thermal storage*). Die Parameter der Kühlgeräte entsprachen den bereits erläuterten



(a)

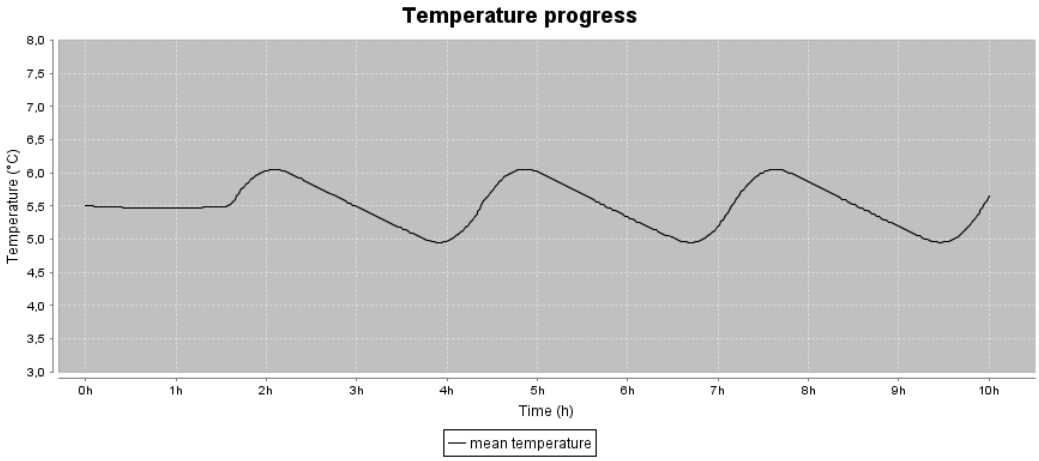


(b)

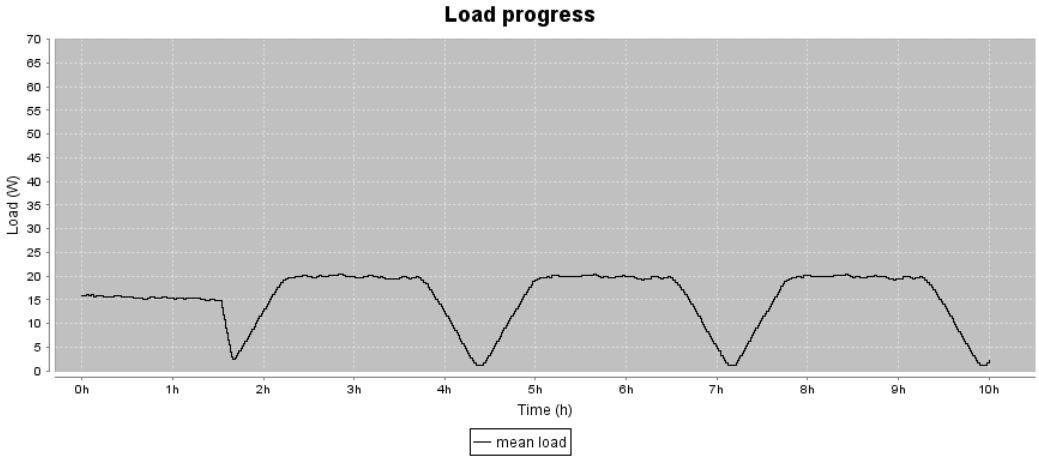
Abbildung 4.4: Auswirkungen des DSC *load* Signales.

Standardwerten, mit Ausnahme der Starttemperatur, die jeweils mittels gleichverteilter Zufallszahlen aus dem Intervall $[T_{min}, T_{max}]$ gesetzt wurde. Zudem starteten 22% der Geräte über $active = true$ in der Phase Kühlen. Das Eintreffen des Signales fand zum Zeitpunkt $t_{notify} = 90\,min$ statt, der Streuungsparameter wurde mit $spread = 10\,min$ angegeben. Es ist deutlich zu erkennen, dass vor dem Eintreffen des Signales eine gleichmäßige Temperatur- und Lastverteilung vorlag. Mit dem Signal steigt die Last dann über einen Zeitraum von 10 Minuten bis zu einem Maximalwert an, um im Anschluss etwas langsamer bis auf den Wert 0 abzusenken. In der Temperaturkurve äußert sich dies durch eine Abkühlung auf einen Minimalwert. Zu diesem Zeitpunkt sind alle Kühlgeräte weitestgehend synchronisiert, d. h. sie schalten nun alle in die Phase Aufwärmen, was sich in der linear steigenden mittleren Temperatur zeigt. Die gemittelte (und hier gleichzeitig absolute) Last beträgt in diesem Zeitraum entsprechend $q = 0\,W$. Da die Geräte nun weiterhin synchronisiert sind, folgen gleichzeitige Wechsel der Phasen, was sich in einer eindeutigen Oszillation der Kurven zeigt. Die Rundungen an den Maximal- und Minimalwerten der Kurven sowie die Tatsache, dass diese Werte nicht den maximal bzw. minimal möglichen Werten entsprechen, liegt an dem zufälligen Schaltverhalten durch den Parameter *spread* sowie den Abweichungen durch die ursprünglich gleichverteilten Temperaturen.
Unter den gegebenen Parameterbelegungen konnte mit diesem Steuerungssignal eine signifikante Reduzierung der Last auf den Minimalwert $q_{min} = 0\,W$ über einen Zeitraum von $\tau_{red} = 85.724\,min$ erfolgen, beginnend zum Zeitpunkt $t_{red} = 136.576\,min$, also knapp 47 Minuten nach Eintritt des Signales.

In Abbildung 4.5 sind die Ergebnisse der gleichen Simulation mit dem komplementären Signal *unload thermal storage* dargestellt. Die Kurvenverläufe verhalten sich erwartungsgemäß in etwa umgekehrt zu den vorherigen Ergebnissen. Auffallend ist die schwächere Ausprägung der Spitzen und Tiefen. Dies liegt in den unterschiedlichen Längen der Zeitspannen $\tau_{cooling}$ und $\tau_{warming}$ begründet. Bei den vorliegenden Parametereinstellungen befinden sich zu jedem Zeitpunkt etwa 22% der Geräte in der Phase Kühlen. Wenn nun ein *load* Signal eintrifft, werden die restlichen 88% zusätzlich in die Phase Kühlen versetzt, was eine starke Veränderung der Kurvenverläufe bewirkt. Beim *unload* Signal ist es umgekehrt, hier befindet sich bereits der größte Teil der Geräte in der angestrebten Phase - und die Auswirkungen in den Kurvenverläufen sind weniger sichtbar.
Genauere Betrachtungen der Parameterabhängigkeiten sowie der möglichen Lastre-



(a)



(b)

Abbildung 4.5: Auswirkungen des DSC *unload* Signales.

duktion sind in [34] gegeben.

4.2 Steuerung nach „Timed Load Reduction“

Die zweite, in [34, Abs.5(2)] angesprochene Art der Steuerung heißt „Timed Load Reduction“ (TLR). Wie der Name schon sagt, bewirkt dieses Signal eine Lastreduzierung, die über mehrere Parameter spezifiziert werden kann. Zunächst einmal gibt es trivialerweise wie beim DSC den Wert t_{notify}, welcher den Zeitpunkt des Eintreffens des Signales beim Controller beschreibt. Das eigentliche Signal beinhaltet dann

zum einen den Parameter $\tau_{preload}$, welcher die Zeitspanne $[t_{notify}, t_{activ}[$ mit $t_{activ} = t_{notify} + \tau_{preload}$ beschreibt, die dem Gerät bis zur erwünschten Lastreduzierung bleibt, um gegebenenfalls den thermischen Speicher zu füllen. Der zweite übergebene Parameter τ_{reduce} beschreibt das Intervall $[t_{notify}+\tau_{preload}, t_{notify}+\tau_{preload}+\tau_{reduce}]$, über das die gewünschte Lastreduzierung erfolgen soll. Optimalerweise füllen also alle Geräte im Intervall $\tau_{preload}$ ihre thermischen Speicher mindestens soweit, dass sie die Spanne τ_{reduce} ohne erneutes Kühlen überstehen können. Allerdings kann es vorkommen, dass ein Kühlgerät dies nicht schafft. Es wurde daher in Anlehnung an [38] eine Klassifizierung vorgenommen, die die Geräte auf Basis ihrer Temperatur $T_{current}$ zum Zeitpunkt t_{notify} nach diesem Kriterium unterscheidet:

- Klasse A
  Die Innentemperatur dieser Geräte ist zum Zeitpunkt t_{notify} so hoch, dass sie es unter ihren jeweiligen Geräteparametern nicht schaffen, während des Zeitraumes $\tau_{preload}$ soweit herunterzukühlen, dass die geforderte Spanne τ_{reduce} ohne erneutes Kühlen überstanden werden kann.

- Klasse B
  Diese Geräte schaffen es, genügend zu kühlen, um τ_{reduce} ohne erneutes Kühlen zu überstehen. Sie sind allerdings nicht in der Lage, im vorgegebenen Zeitraum $\tau_{preload}$ ihre Minimaltemperatur T_{min} zu erreichen.

- Klasse B1
  Die Geräte in dieser Klasse sind zum Zeitpunkt t_{notify} bereits soweit herabgekühlt, dass eine weitere Kühlung nicht notwendig ist. Sie schaffen es, den geforderten Zeitraum ohne Kühlen zu überstehen, selbst wenn sie zum Zeitpunkt t_{notify} bereits in die Phase Aufwärmen schalten. Wie die Geräte aus der Klasse B sind sie jedoch nicht in der Lage, T_{min} zu erreichen.

- Klasse C
  Die Temperatur und Parameter dieser Geräte erlauben es ebenfalls, im Zeitraum $\tau_{preload}$ genügend zu kühlen. Zusätzlich schaffen diese Geräte es, in diesem Zeitraum die Minimaltemperatur T_{min} zu erreichen.

- Klasse C1
  Wie die Geräte in der Klasse B1 sind diese Geräte zum Zeitpunkt t_{notify} bereits soweit herabgekühlt, dass eine weitere Kühlung nicht notwendig ist. Ähnlich den Geräten aus der Klasse C sind sie außerdem zusätzlich in der Lage, T_{min} zu erreichen.

- Klasse D
  Die Geräte in dieser Klasse sind durch den großen zeitlichen Abstand von t_{notify} bis zum Aktivierungszeitpunkt t_{activ} nicht in der Lage, durch eine *unmittelbare* Modifikation des regulären Kühlprogrammes (vgl. Abschnitt 3.3) die erforderliche Grenztemperatur $T_{max-act}$ zum Aktivierungszeitpunkt zu erreichen.
  Bei einer sofortigen Kühlung beispielsweise würden sie die minimale Temperatur T_{min} zu früh erreichen und die Temperatur würde ohne weitere Anpassung des weiteren Verhaltens durch die nachfolgende Aufwärmphase noch während des Intervalls $\tau_{preload}$ über die Grenztemperatur $T_{max-act}$ steigen.

Im Folgenden wird ersichtlich werden, dass nicht alle dieser Klassen für einzelne Steuerungsstrategien des Intervalls $\tau_{preload}$ notwendig sind. Doch zunächst sollen die Klassen und ihre Kennwerte genauer beschrieben und formalisiert werden. Abbildung 4.6 visualisiert die gesamte Klasseneinteilung anhand einer Bereichskarte der Klassen im Temperaturverlauf des betreffenden Zeitintervalls um den Zeitpunkt des Steuerungssignales. Wird nun ein Steuerungssignal zu einem beliebigen Zeitpunkt

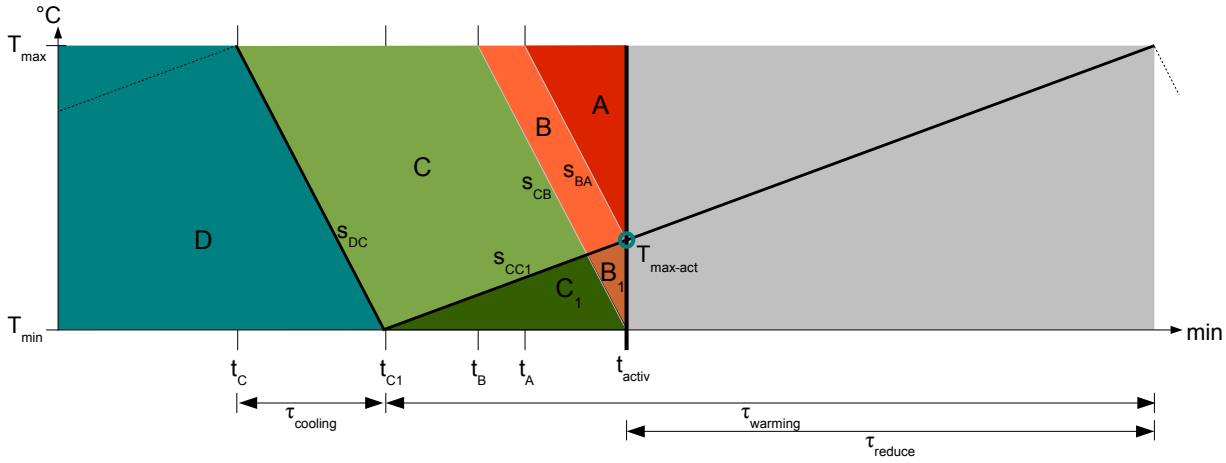


Abbildung 4.6: Bereichskarte der TLR Klasseneinteilung.

t_{notify} initiiert, so lässt sich anhand der Innentemperatur T_{notify} eines Kühlgerätes zu diesem Zeitpunkt von der Karte ablesen, ob das Gerät in die Klasse A, B, B_1, C, C_1 oder D fällt. Mit beispielsweise $t_{notify} > t_A$ und $T_{notify} = T_{max}$ würde ein Gerät mit A klassifiziert werden.
Der mit einem Kreis markierte Punkt in dem Diagramm zeigt die maximale Tem-

peratur $T_{max-act}$, die ein Kühlgerät zum Aktivierungszeitpunkt t_{activ} haben kann, damit es das Reduktionsintervall übersteht. Die schwarze fettgedruckte Kurve beschreibt den Temperaturverlauf eines Kühlgerätes, das exakt diese Bedingung erfüllt. Unterhalb des Diagrammes sind zur Orientierung die Zeitspannen $\tau_{cooling}$, $\tau_{warming}$ und τ_{reduce} aufgetragen. Die Geraden, welche die einzelnen Bereiche innerhalb des Diagrammes voneinander trennen, sind mit s_{XY} bezeichnet (*„s trennt die Bereiche X und Y“*). Die markierten Zeitpunkte t_X bezeichnen den Beginn des jeweiligen Zeitintervalls, in dem ein Kühlgerät in die Klasse X fallen kann. Ausgehend von diesen Eckdaten kann die Klassifizierung eines Kühlgerätes nun wie folgt formalisiert werden. Für die Zeitpunkte t_X gilt:

$$
\begin{aligned}
t_A &= t_{activ} - \frac{T_{max-act} - T_{max}}{a_c}\,, && (4.1)\\
t_B &= t_{activ} - \tau_{cooling}\,, && (4.2)\\
t_C &= t_{activ} + \tau_{reduce} - \tau_{warming} - \tau_{cooling}\,, && (4.3)\\
t_{C1} &= t_{activ} + \tau_{reduce} - \tau_{warming}\,. && (4.4)
\end{aligned}
$$

Ausgehend von diesen Werten lassen sich die Funktionsgleichungen der Trenngeraden definieren:

$$
\begin{aligned}
s_{DC}(t) &= T_{max} + a_c \cdot (t - t_C)\,, && (4.5)\\
s_{CC1}(t) &= T_{min} + a_w \cdot (t - t_{C1})\,, && (4.6)\\
s_{CB}(t) &= T_{max} + a_c \cdot (t - t_B)\,, && (4.7)\\
s_{BA}(t) &= T_{max} + a_c \cdot (t - t_A)\,. && (4.8)
\end{aligned}
$$

Der Wert $T_{max-act}$ für Formel 4.1 kann dabei durch $s_{CC1}(t_{activ})$ berechnet werden. Die Bedingungen für die Klassifizierungen sind dann schlussendlich durch die folgenden Formeln gegeben:

$$A \Leftrightarrow T_t > s_{BA}(t) , \tag{4.9}$$
$$B \Leftrightarrow T_t > s_{CB}(t) \wedge T_t \leq s_{BA}(t) \wedge T_t > s_{CC1}(t) , \tag{4.10}$$
$$B_1 \Leftrightarrow T_t \leq s_{CC1}(t) \wedge T_t > s_{CB}(t) , \tag{4.11}$$
$$C \Leftrightarrow T_t \geq s_{DC}(t) \wedge T_t > s_{CC1}(t) \wedge T_t \leq s_{CB}(t) , \tag{4.12}$$
$$C_1 \Leftrightarrow T_t \leq s_{CC1}(t) \wedge T_t \leq s_{CB}(t) , \tag{4.13}$$
$$D \Leftrightarrow T_t < s_{DC}(t) . \tag{4.14}$$

Dabei bezeichnet t den Zeitpunkt, zu dem die Klassifizierung vorgenommen werden soll, und T_t ist die Temperatur des Kühlgerätes zu diesem Zeitpunkt.

Ein Sonderfall ergibt sich, wenn das Reduktionsintervall größer als die technisch mögliche Dauer der Aufwärmphase ist ($\tau_{reduce} > \tau_{warming}$). Unter solchen Umständen kann obige Klassifizierung nicht vorgenommen werden. Eine Bereichskarte, die diesen Fall beschreibt, ist in Abbildung 4.7 gegeben.

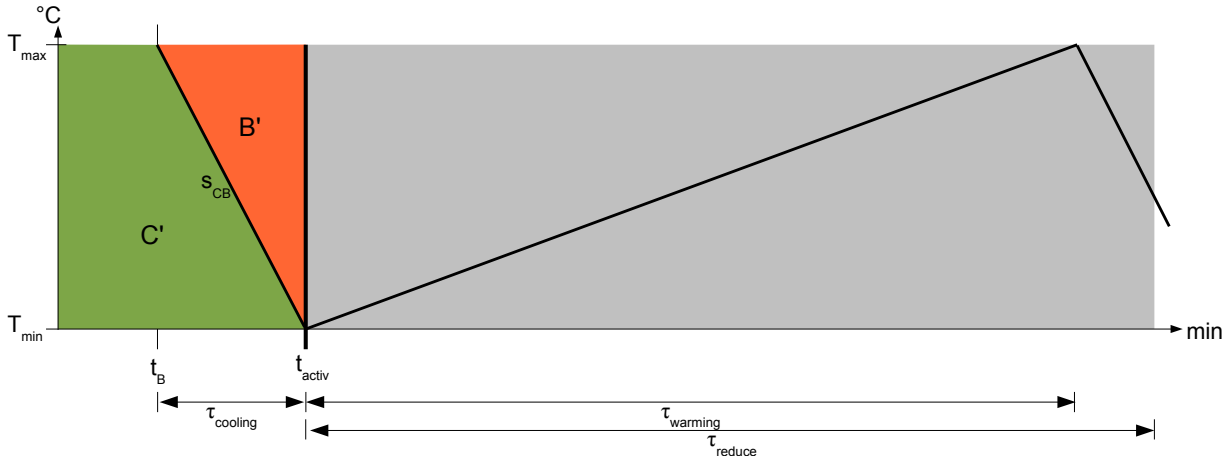


Abbildung 4.7: Bereichskarte für $\tau_{reduce} > \tau_{warming}$.

Es existieren nur noch die Klassen B' und C' in Anlehnung an B und C. Diese sind wie folgt definiert:

- Klasse B'
  Die Geräte dieser Klasse sind nicht in der Lage, bis zum Start des Reduktionsintervalles in t_{activ} ihre Minimaltemperatur T_{min} zu erreichen.
- Klasse C'
  Die Geräte dieser Klasse sind im Gegensatz dazu in der Lage, bis zum Start des Reduktionsintervalles in t_{activ} ihre Minimaltemperatur T_{min} zu erreichen.

Mathematisch ausgedrückt werden die beiden Klassen durch folgende Formeln definiert:

$$B' \quad \Leftrightarrow \quad T_t > s_{CB}(t) \ , \tag{4.15}$$

$$C' \quad \Leftrightarrow \quad T_t \leq s_{CB}(t) \ . \tag{4.16}$$

Die Trenngerade s_{CB} wurde unverändert übernommen, ihre Gleichung ist weiterhin gültig. Gleiches gilt für den Zeitpunkt t_B, zu welchem der Bereich B' beginnt.

Es ist für beide Fälle leicht einzusehen, dass das Steuerungspotenzial der Geräte mit sinkender Temperatur T_{notify} zum Zeitpunkt $t = t_{notify}$ zunimmt, da eine niedrige Ausgangstemperatur weniger zusätzliche Kühlung im Zeitraum $\tau_{preload}$ erfordert. Basierend auf den hier definierten Klassen, Zeitpunkten und Trenngeraden können nun diverse Steuerungsstrategien für das Intervall $\tau_{preload}$ entwickelt werden, die die Kühlgeräte nach bestimmten Anforderungen umprogrammieren. So wäre etwa eine einfache Strategie denkbar, die sofort zu Beginn des Intervalls alle Kühlgeräte soweit wie möglich herunterkühlt, und zum Zeitpunkt t_{activ} alle Geräte in die Phase Aufwärmen schaltet. Hier sind nur wenig Berechnungen erforderlich, allerdings kann diese Strategie unter bestimmten Umständen nicht die optimal mögliche Reduktionsdauer gewährleisten (etwa weil das sofortige Kühlen eine Umschaltung noch früh im Preload-Intervall in die Aufwärmphase der Geräte bewirkt, wodurch sie nicht mehr in der Lage sind, das Reduktionsintervall weit genug zu überstehen). Eine etwas komplexere Strategie wäre es beispielsweise, für die Kühlgeräte genau solch ein Kühlprogramm zu berechnen, dass sie zum Zeitpunkt t_{activ} eine möglichst niedrige Temperatur haben und das Reduktionsintervall (und eventuell einige Zeit darüberhinaus) möglichst lang ohne Kühlung überstehen.
Im folgenden Abschnitt wird die konkrete Implementierung einer Strategie aus [34] beschrieben. Diese Strategie versucht, das Kühlprogramm der Geräte soweit anzupassen, dass sie das Reduktionsintervall *optimal* überstehen, also nach Möglichkeit

genau zum Zeitpunkt t_{activ} stoppen zu kühlen, und direkt nach Beendigung des Reduktionsintervalls wieder beginnen zu kühlen. Dies wird durch eine Ansteuerung der Zieltemperatur $T_{max-act}$ versucht zu erreichen.

4.2.1 Implementierung TLR

Um die Steuerung TLR in der bereits implementierten Basisversion des Modells zu realisieren, wurde wie bei der Implementierung von DSC zunächst ein Interface ItIr angelegt, welches das Steuerungssignal repräsentiert. Listing 4.2 zeigt den Inhalt des Interfaces.

Listing 4.2: Interface ItIr

```
1 public interface ItIr {
2   public final static String EV_REDUCE_LOAD = "ReduceLoad";
3
4   public abstract void doReduceLoad(Double tau_preload,
5                                     Double tau_reduce);
6 }
```

Abbildung 4.8 zeigt den Ereignisgraphen der um das TLR-Signal erweiterten Version des Controllers für die bereits beschriebene Strategie des optimalen Kühlens im Intervall $\tau_{preload}$, um das Reduktionsintervall τ_{reduce} exakt zu überstehen. Für andere Steuerungsstrategien würde der Graph demnach leicht abweichend (und in vielen Fällen nicht so komplex) aussehen. Die Abbildung zeigt die Anbindung zum linearen Controller (kompakte Version). Da die iterative und die nichtkompakte lineare Variante fast identisch aufgebaut sind werden sie hier nicht weiter erläutert. Zudem wurden die Operationen der Ereignisse entfernt, um die Übersichtlichkeit zu erhöhen. Das stellt kein Problem dar, da die Operationen nichts weiter bewirken, als die Parameter, Verzögerungen und Bedingungen der ausgehenden Kanten der jeweiligen Ereignisse zu berechnen. Unten in der Abbildung ist das vom Interface ItIr vorgeschriebene Ereignis ReduceLoad zu erkennen. Von diesem Ereignis aus führt eine Fallunterscheidung auf Basis der weiter oben eingeführten Klassifizierung über eines der drei Folgeereignisse Intersect s_{CC1}, Intersect s_{CB} und Intersect s_{BA} zu einer Modifikation des regulären Kühlprogrammes (vgl. Abschnitt 3.3) im Ereignis DeleteAndTargetTo (vgl. Abschnitt 4.1.1).
Tritt nun das Ereignis ReduceLoad ein, so wird zunächst ermittelt, ob das entsprechende Kühlgerät technisch in der Lage ist, das geforderte Reduktionsintervall

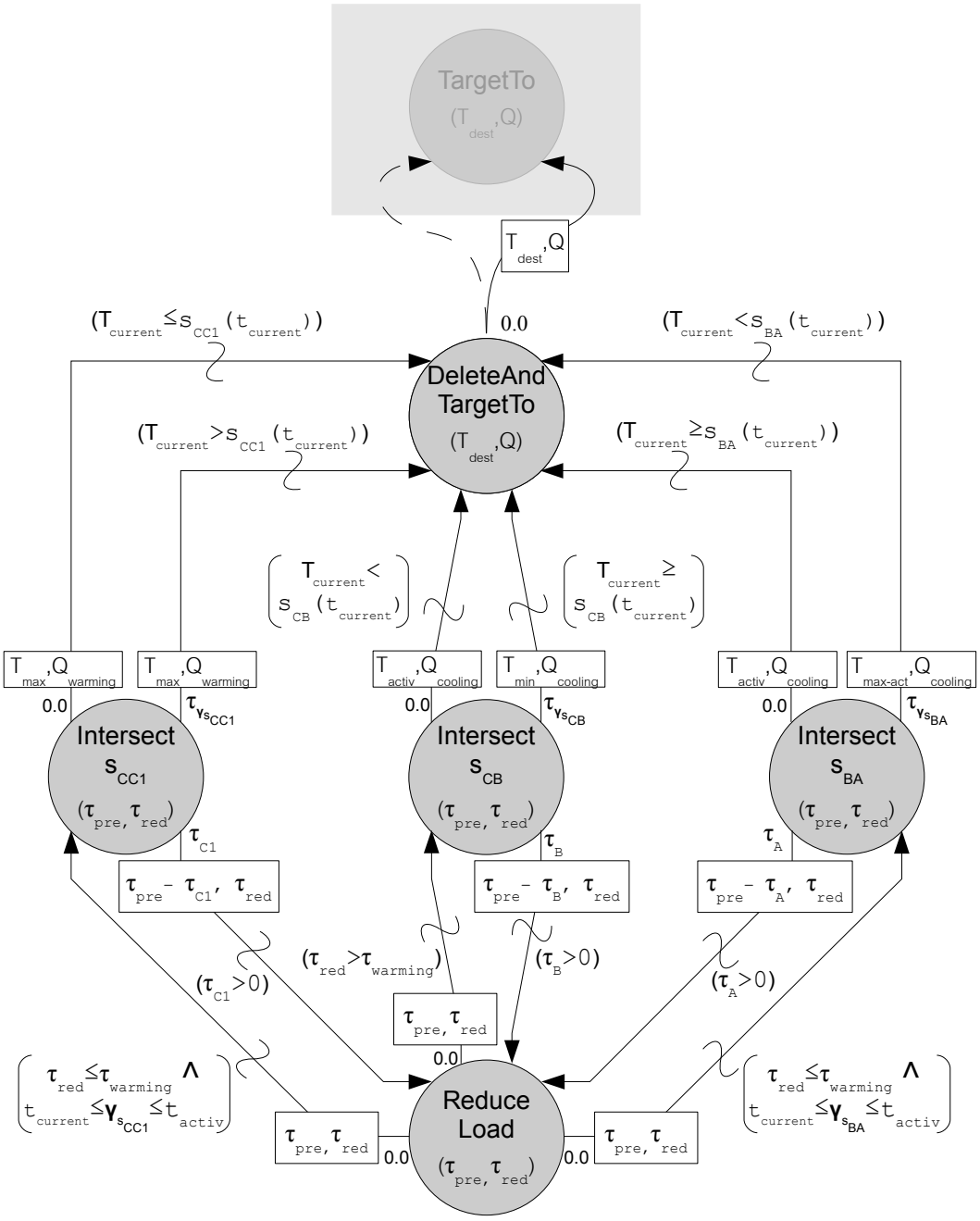


Abbildung 4.8: Kompakter Ereignisgraph des linearen TLR-Controllers.

zu überstehen. Falls nicht, bedeutet dies, dass das gewünschte Intervall länger als die vollständige Aufwärmdauer des Gerätes ist ($\tau_{reduce} > \tau_{warming}$), es gilt also $T_{max-act} < T_{min}$. In diesem Fall wird das beste Ergebnis erzielt, wenn das Gerät mit seiner Minimaltemperatur T_{min} in das Reduktionsintervall einsteigt, da es so die Reduktion weitestmöglich erfüllt. In dem dargestellten Ereignisgraphen wird dieser Fall durch den mittleren Weg, also durch das das Ereignis Intersect s_{CB} beschrieben.

Intersect s_{CB}:

In diesem Ereignis wird der aktuelle Temperaturverlauf des Kühlgerätes mit der Trenngeraden s_{CB} verglichen. Falls die Temperatur des Gerätes zum diesem Zeitpunkt bereits über der Trenngeraden liegt ($T_{current} > s_{CB}(t_{current})$, siehe Formel 4.7) so kann T_{min} bis zum Zeitpunkt t_{activ} nicht mehr erreicht werden (dies entspricht der Klasse B'). Es wird daher versucht, soweit wie möglich herunterzukühlen, um das Reduktionsintervall dennoch möglichst lange zu überstehen. Dies wird durch die Berechnung der Temperatur T_{activ} erreicht, die das Gerät bis t_{activ} erreichen kann. Diese Temperatur wird dann als Zieltemperatur an das Modifikationsereignis DeleteAndTargetTo übergeben, welches das aktuelle Kühlprogramm dem Parameter entsprechend anpasst. Liegt die Temperatur unter der Trenngeraden (Klasse C'), so ist das Gerät in der Lage, seine Minimaltemperatur zu erlangen. Damit es diese Temperatur exakt zum Zeitpunkt t_{activ} erreicht (und damit die längstmögliche Reduktion), wird mittels linearer Berechnungen der Schnittpunkt des aktuellen Kühlprogrammes mit der Geraden s_{CB} bestimmt. Der Abszissenwert $\gamma_{s_{CB}}$ dieses Punktes beschreibt den Zeitpunkt, zu dem das Gerät in die Kühlphase schalten muss, um mit seiner minimal möglichen Temperatur in das Reduktionsintervall einzusteigen. Der Abstand zu diesem Zeitpunkt wird mittels $\tau_{\gamma_{s_{CB}}} = \gamma_{s_{CB}} - t_{current}$ berechnet und als Verzögerung für das Modifikationsereignis DeleteAndTargetTo verwendet. Diesem Ereignis wird dann die Zieltemperatur T_{min} übergeben und somit das gewünschte Verhalten erzielt.

Ist das Gerät allerdings technisch in der Lage, die geforderte Reduktionsdauer zu überstehen ($\tau_{reduce} \leq \tau_{warming}$), so wird statt des soeben beschriebenen Ansteuerns der Minimaltemperatur versucht, exakt die Grenztemperatur $T_{max-act}$ im Zeitpunkt t_{activ} zu erreichen. In Bezug auf Abbildung 4.6 wird also der Schnittpunkt mit einer der Geraden s_{CC1} und s_{BA} berechnet (linker bzw. rechter Weg im Ereignisgraphen), um ausgehend davon das Kühlprogramm auf die Temperatur $T_{max-act}$ hin anzupassen. Welcher der Wege verwendet wird hängt davon ab, welcher von den beiden Schnittpunkten im Intervall $[t_{current}, t_{activ}]$ sowie im Bereich $[T_{min}, T_{max}]$ liegt. Falls dies für beide zutrifft, wird derjenige verwendet, dessen Ab-

szissenwert ($\gamma_{s_{CC1}}$ bzw. $\gamma_{s_{BA}}$) näher am aktuellen Zeitpunkt $t_{current}$ liegt.

Intersect s_{CC1}:

Im Falle von $\gamma_{s_{CC1}} < \gamma_{s_{BA}}$ (linker Weg) muss zudem überprüft werden, ob sich das Gerät bereits im Bereich C1 oder B1 befindet, denn dann kann die Temperatur $T_{max-act}$ bis zum Zeitpunkt t_{activ} nicht mehr erreicht werden. Dies stellt kein großes Problem dar, da das Reduktionsintervall trotzdem überstanden wird, wenn auch etwas länger als bei dieser Strategie erwünscht. In diesem Fall ($T_{current} \leq s_{CC1}(t_{current})$) wird das Gerät unmittelbar in den Aufwärmbetrieb geschaltet, falls es noch nicht in dieser Phase ist. Im anderen Fall ($T_{current} > s_{CC1}(t_{current})$) wird ähnlich wie beim mittleren Weg der Abstand $\tau_{\gamma_{s_{CC1}}} = \gamma_{s_{CC1}} - t_{current}$ als Verzögerung für die Umschaltung in den Aufwärmbetrieb verwendet, um so exakt den Punkt $(t_{activ}, T_{max-act})$ zu durchlaufen.

Intersect s_{BA}:

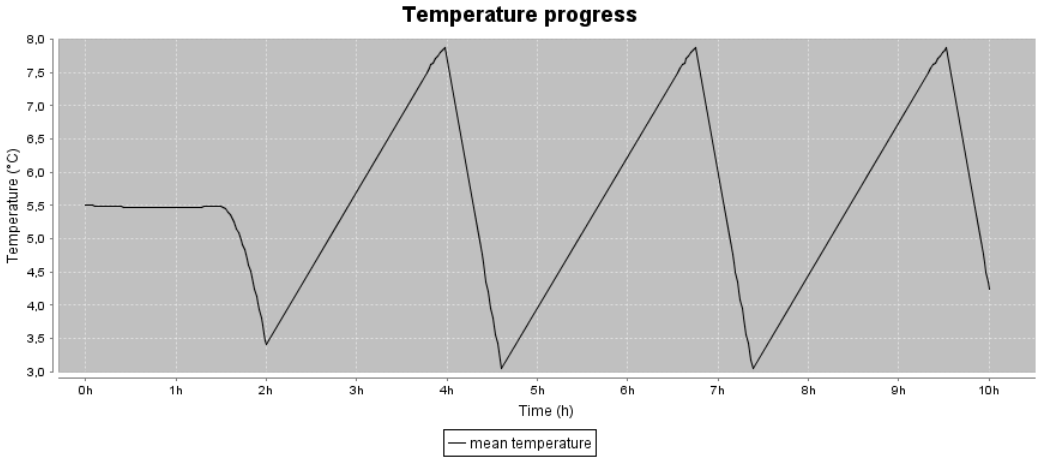
Falls andererseits der Schnittpunkt mit der Trenngeraden s_{BA} zuerst eintritt, und also $\gamma_{s_{BA}} < \gamma_{s_{CC1}}$ gilt, wird der rechte Weg im Ereignisgraphen über das Ereignis Intersect s_{BA} beschritten. Auch hier wird wieder zwischen dem Bereich „über“ der Geraden ($T_{current} \geq s_{CC1}(t_{current})$, Klasse A) und „unter“ der Geraden ($T_{current} < s_{BA}(t_{current})$, Klassen B, C und D) unterschieden. Für A gilt, dass das Gerät die Temperatur $T_{max-act}$ im Zeitpunkt t_{activ} nicht mehr erreichen kann, da die aktuelle Temperatur zu hoch ist und das Gerät nicht mehr genügen Zeit hat um herunterzukühlen. Das Kühlprogramm wird in diesem Fall so modfiziert, dass ähnlich wie im Ereignis Intersect s_{CB} die Temperatur T_{activ} berechnet wird, die das Gerät bis t_{activ} erreichen kann, und diese dann als Zieltemperatur an DeleteAndTargetTo übergeben wird, um so eine maximale Reduktionsdauer zu erreichen. Im anderen Fall, wenn die Temperatur unter der Geraden liegt, wird wiederum der Abstand zum Schnittpunkt mit der Geraden durch $\tau_{\gamma_{s_{BA}}} = \gamma_{s_{BA}} - t_{current}$ berechnet und dieser als Verzögerung für das Modifikationsereignis verwendet, um genau $T_{max-act}$ in t_{activ} zu erreichen.

Zusätzlich besitzen die drei soeben beschriebenen Ereignisse jeweils eine rücklaufende Kante, die erneut zum Ereignis ReduceLoad führt. Diese Kanten werden genau dann verfolgt, wenn die obigen Berechnungen nicht möglich sind, weil der engere Bereich um die betreffenden Zeitpunkte und Geradensegmente noch nicht erreicht wurde. Diese Kanten sind sozusagen eine Sicherheit für den Fall, dass $\tau_{preload}$ zu großzügig gewählt wurde, sie verschieben die Operation des Controllers in den unmittelbaren Einzugsbereich der Trenngeraden. Dabei bezeichnen die Verzögerungswerte τ_B, τ_{C1} und τ_A an den Kanten genau die Abstände zu den Zeitpunkten t_B,

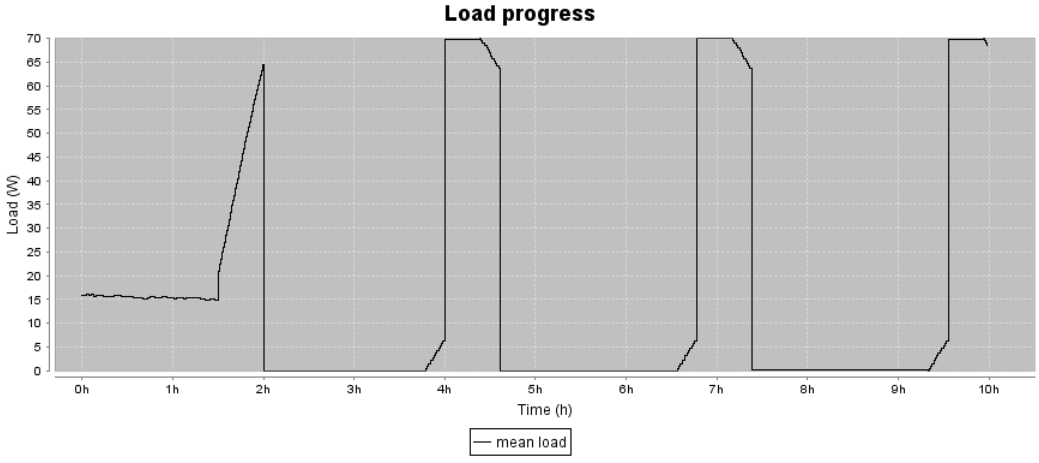
t_{C1} und t_A, zu denen die jeweiligen Einzugsbereiche B, C1 und A beginnen (vgl. Abbildung 4.6 und 4.7).

4.2.2 Simulationsergebnisse nach TLR-Steuerung

Abbildung 4.9 zeigt die Ergebnisse einer Simulation von 5000 Kühlgeräten mit der soeben beschriebenen kompakten Variante des linearen Controllers und aktiviertem TLR-Signal. Die Signalparameter wurden mit $t_{notify} = 90\,min$, $\tau_{preload} = 30\,min$



(a)



(b)

Abbildung 4.9: Auswirkungen des TLR-Signales.

und $\tau_{reduce} = 120\,min$ initialisiert, die Simulationsdauer betrug wie in den vorher-

gehenden Beispielen 10 Stunden. Die Geräteparameter entsprechen den Standardeinstellungen aus den vorigen Kapiteln. Da mit diesem Parametern für alle Kühlgeräte gilt

$$\tau_{warming} \approx 129.882min < \tau_{reduce}\,,$$

wird in dieser Simulation niemals im Ereignisgraphen der Weg über Intersect s_{CB} gegangen, es erfolgt also grundsätzlich der Versuch einer Ansteuerung der Temperatur $T_{max-act}$ zum Beginn des Reduktionsintervalles. Diejenigen Geräte, die zum Zeitpunkt t_{notify} bereits so warm sind, dass sie $T_{max-act}$ nicht mehr erreichen können, müssen sofort beginnen zu kühlen, um dennoch eine möglichst niedrige Einstiegstemperatur in das Reduktionsintervall zu erlangen. Dies ist in der Abbildung in dem unmittelbaren Anstieg der mittleren Last zum Zeitpunkt $t_{notify} = 90\,min$ um ca. $7\,W$ erkennbar. Im Anschluss steigt die Last gleichmäßig an, bis zu einem Maximum zum Zeitpunkt $t_{activ} = t_{notify} + \tau_{preload} = 120\,min$, in welchem das Reduktionsintervall startet und alle noch aktiven Geräte in den Aufwärmbetrieb schalten. Der gleichmäßige Anstieg kann dadurch erklärt werden, dass die Temperaturen der Geräte gleichverteilt sind und die zur Zeit passiven Geräte nach und nach die Gerade s_{BA} schneiden und beginnen zu kühlen, damit sie $T_{max-act}$ erreichen. Die restlichen Geräte, für die die beiden Fälle nicht zutreffen, wurden bereits vor t_{activ} in den Aufwärmbetrieb geschaltet. Diese Gruppe repräsentiert den Weg über Intersect s_{CC1} im Ereignisgraphen und ist in der Lastkurve daran zu erkennen, dass das Maximum in t_{activ} nur bei ca. $64\,W$ statt $70\,W$ liegt.
Die nun folgende Reduktionsphase kann also von allen Geräten überstanden werden, deren Temperatur sich in $\tau_{preload}$ unterhalb der Geraden s_{BA} befand. Diejenigen, für die dies nicht gilt, und die den unmittelbaren Anstieg der Last in t_{notify} verursachten, müssen noch vor dem Ende von τ_{reduce} wieder beginnen zu kühlen. Der graduelle Anstieg der Last im Diagramm bis auf etwa $7\,W$ im Ende des Intervalles visualisiert diese Gerätegruppe. Der Steuerungsstrategie nach schalten nun folgend alle übrigen Geräte gleichzeitig in den Kühlbetrieb; die Last steigt auf den maximal möglichen Wert $q_{max} = 70\,W$. Diese Geräte sind vollständig synchronisiert und werden bis auf die eben erwähnte Gruppe alle für den gleichen Zeitraum $\tau_{cooling}$ in dieser Phase bleiben. Die Grafik verdeutlicht dies durch den Abfall der Last nach $\tau_{cooling} \approx 36.645\,min$ auf den Minimalwert $q_{min} = 0\,W$. Die leichte graduelle Senkung der Maximallast auf $70\,W - 7\,W$ unmittelbar vor diesem Zeitpunkt ist wiederum durch die obige Gruppe der „Nicht-Überbrücker“ zu erklären. Dieses Verhalten wiederholt sich nun jeweils im Abstand von $\tau_{warming} + \tau_{cooling} \approx 166.527\,min$.

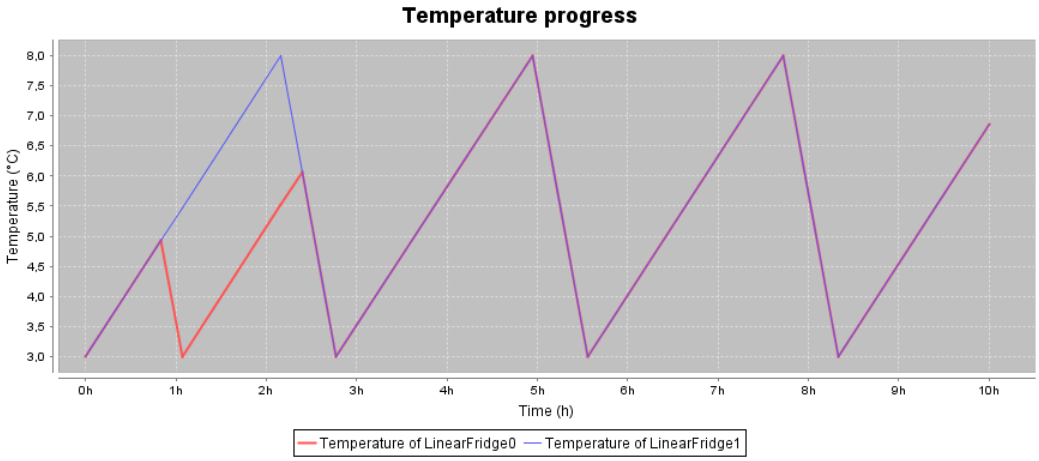
Kapitel 5

Dämpfungsstrategien

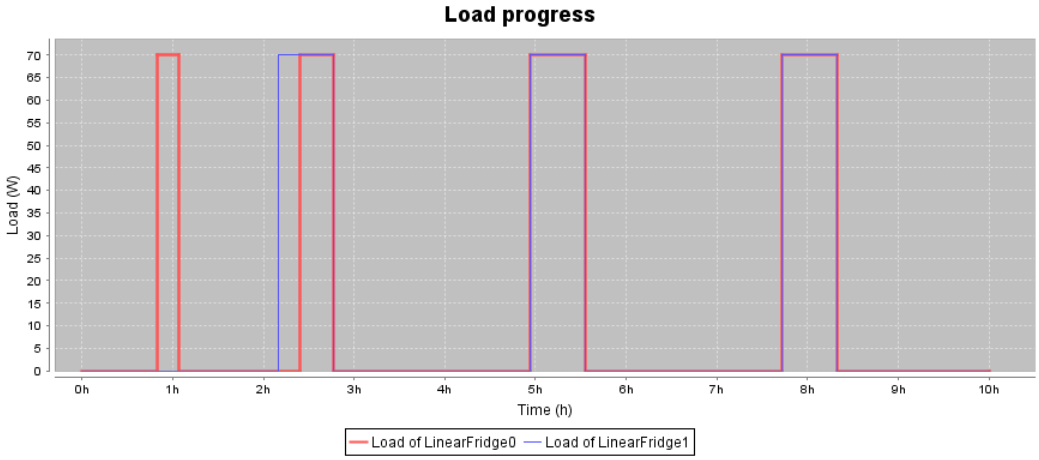
Die in dem vorhergehenden Kapitel vorgestellten Steuerungssignale DSC und TLR sollen nun im Folgenden auf Möglichkeiten zur Dämpfung der durch die jeweilige Steuerung entstehende Oszillation von synchronen Hochlast- und Tieflastphasen untersucht werden.

5.1 Zustandswiederherstellung

Da eine Modifikation des Kühlprogrammes nach DSC oder TLR lediglich die Schaltpunkte der Phasen Kühlen und Aufwärmen verändert, lässt sich der Zustand eines Kühlgerätes als Tupel $(Temperatur, Phase, Last)$ auffassen. Eine Strategie zur Wiederherstellung einer gleichverteilten Population von Gerätezuständen wäre daher, nach einem erfolgten Steuerungssignal genau die Zustandsverteilung anzusteuern, die die Gerätepopulation haben würde, wenn die Steuerung nicht statt gefunden hätte. Abbildung 5.1 demonstriert diese Strategie am Beispiel eines einzelnen Kühlgerätes. Die dünnere blaue Kurve („LinearFridge1“) zeigt jeweils den Temperatur- und Lastverlauf eines Gerätes ohne externe Steuerung. Die etwas dickere rote Kurve („LinearFridge0“) beschreibt den Verlauf des gleichen Gerätes, wenn es etwa zum Zeitpunkt $t_{notify} = 50\,min$ ein DSC-*load* Signal erhält. Das Signal bewirkt, dass das Gerät unmittelbar auf seine Minimaltemperatur herunterkühlt. Die darauf folgenden Schaltpunkte der Phasen sind nun jedoch verschoben im Vergleich zum unmodifizierten Gerät, wodurch die bereits erläuterte Synchronisation entsteht. Es ist nun ein Algorithmus denkbar, der nach dem Steuerungssignal und seinem Effekt (in diesem Fall also nach dem Abkühlen auf die Minimaltemperatur) eine weitere Modifikation berechnet, die das Gerät wieder in seinen ursprünglichen Rhythmus versetzt.



(a)



(b)

Abbildung 5.1: Strategie: Zustandswiederherstellung nach DSC-Signal.

Diese Modifikation ist in dem Diagramm etwa zum Zeitpunkt $t_{restore} = 140\,min$ zu erkennen. Grafisch betrachtet trifft dort die Temperaturkurve des modifizierten Gerätes auf die Temperaturkurve der unmodifizierten Variante. Doch statt sie zu schneiden, wird durch die zweite Modifikation erreicht, dass ab sofort wieder dem ursprünglichen Verlauf gefolgt wird. Es wird also genau der Zustand hergestellt, den das Gerät ohne eine Modifikation gehabt hätte. Wendet man dieses Prinzip auf eine große Population von Geräten an, so müsste sich eine vollständige Desynchronisierung einstellen.

Die Implementierung der Zustandswiederherstellung kann durch Vererbung der Klassen als Erweiterung zu einem beliebigen Controller realisiert werden. Listing 5.1 zeigt das dazugehörige Interface, über welches die Funktionalität der Zustandswiederherstellung definiert wird. Enthalten ist lediglich die Vorgabe eines Ereignisses,

Listing 5.1: Interface IStateful

```
1 public interface IStateful {
2
3   public final static String EV_RESTORE_STATE = "RestoreState";
4
5   public abstract void doRestoreState();
6 }
```

das bei Eintritt einen zuvor gesicherten Zustand wiederherstellt. Die Realisierung dieser Erweiterung gestaltet sich allerdings etwas schwieriger als beispielsweise die Implementierung des DSC-Signals als Erweiterung zum Basiscontroller (siehe Abschnitt 4.1.1), da es nicht ausreicht, zusätzliche Ereignisse zu definieren, die bei Bedarf eingeplant werden. Zusätzlich müssen die bestehenden Ereignisse angepasst werden. In Bezug auf Abbildung 4.3 müssten etwa die Ereignisse LoadThermalStorage und UnloadThermalStorage um das Speichern des Signaltyps (*load* bzw. *unload*) erweitert werden. Im Ereignis DeleteAndTargetTo muss dann direkt vor der Modifikation berechnet werden, wann sich der unmodifizierte und der modifizierte Temperaturverlauf wieder schneiden würden. Zudem muss der virtuelle Zustand des Gerätes in diesem Zeitpunkt berechnet und als Zielzustand für die Zustandswiederherstellung gespeichert werden. Letztendlich muss dann das Ereignis RestoreState für eben diesen Zeitpunkt eingeplant werden, bevor die eigentliche Modifikation durchgeführt wird. Das später eingeplante Ereignis RestoreState braucht dann nichts anderes mehr zu tun, als bei Eintritt das Gerät in die in dem gesicherten Zustand enthaltene Phase mit der angegebenen Last zu schalten, um so das Gerät wieder seinem ursprünglichen Rhythmus folgen zu lassen. Abbildung 5.2 zeigt den resultierenden Ereignisgraphen für den linearen Controller (kompakte Variante). Die Operationen an den Ereignissen bewirken das soeben beschriebene Verhalten, indem sie die notwendigen Berechnungen durchführen. Die Namen der Operationen sind exemplarisch, die Anweisung calculateIntersection() im Ereignis DeleteAndTargetTo steht beispielsweise für den Algorithmus, der den Zeitpunkt berechnet, zu dem die modifizierte Temperaturkurve das nächste mal auf die ursprüngliche Kurve trifft. Die Arbeitsweise der Anweisung ist Listing 5.2 zu entnehmen. Das Objekt fridge bezeichnet dabei das Kühlgerät,

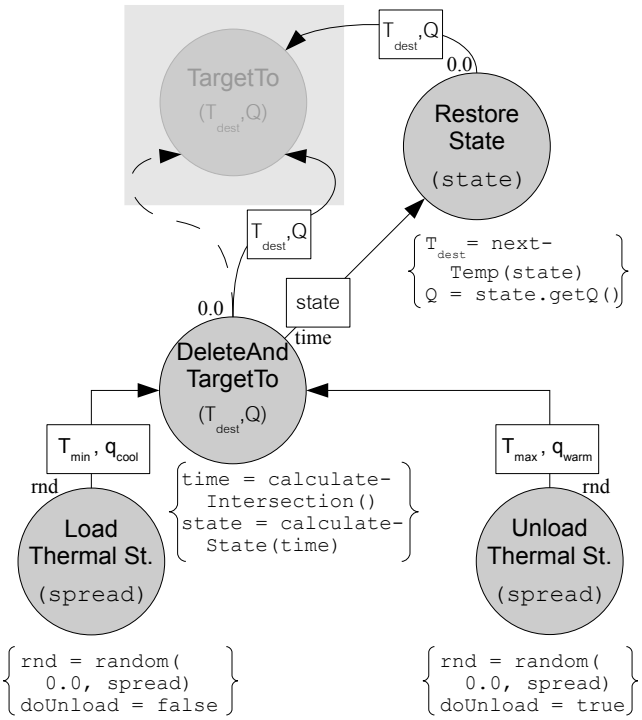


Abbildung 5.2: Linearer DSC-Controller (kompakt) mit Zustandswiederherstellung.

dem der Controller zugeordnet ist. Die äußeren if-Bedingungen bewirken, dass nur dann eine zweite Modifikation vorgenommen wird, falls überhaupt eine erste Modifikation stattfindet. Denn wenn das Signal load erfolgt, und sich das Gerät bereits in der Phase Kühlen befindet, so ändert sich nichts an seinem Verhalten, und der Zustand muss später nicht wiederhergestellt werden. Ebenso verhält es sich, wenn das Signal unload anliegt, und sich das Gerät in der Phase Aufwärmen befindet. Innerhalb der Bedingungen wird dann zunächst auf Basis der aktuellen Temperatur die Temperatur berechnet, zu der sich die beiden Kurven das erste mal schneiden. Mittels der Methode fridge.tau(...) wird dann die Zeitspanne ermittelt, nach der das Gerät ebendiese Temperatur und damit den Schnittpunkt erreicht. Die Kommentare vor den jeweiligen Aufrufen verdeutlichen dies noch einmal direkt am Code.
Die folgende Operation calculateState(time) bezeichnet den Algorithmus, der dann den fiktiven Zustand des unmodifizierten Gerätes zu genau diesem Zeitpunkt ermittelt. Die Funktionsweise ist im vorliegenden Beispiel ähnlich zu der soeben beschriebenen Operation: Es wird zunächst die Temperatur berechnet, zu der sich die

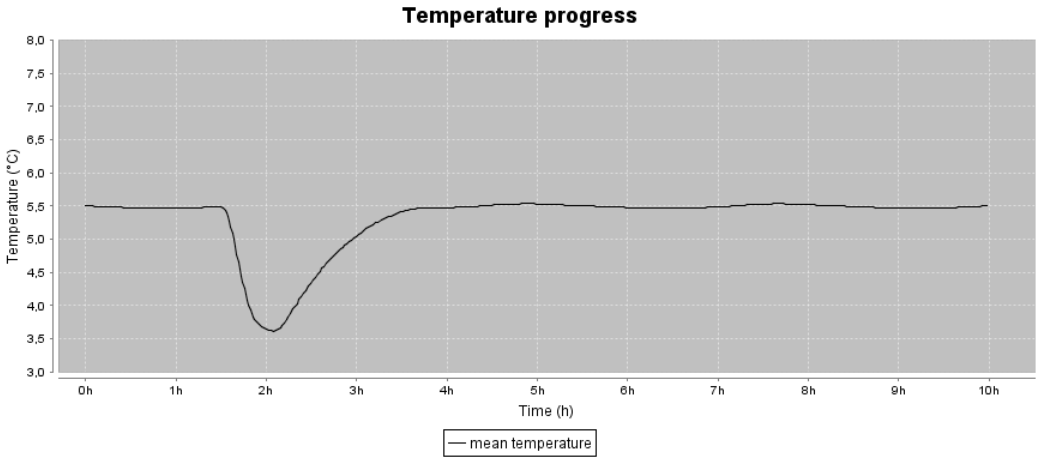
Listing 5.2: Anweisung calculateIntersection()

```
double calculateIntersection() {
  if (!doUnload && !fridge.isActive()) {
    double temp = fridge.getT_current();
    // T_crossing = temperature where regular and modified will cross
    // the first time
    double T_crossing = fridge.getT_min() + fridge.getT_max() - temp;
    // timespan to this first crossing = time to cool down to T_min ...
    return fridge.tau(temp, fridge.getT_min(), fridge.getQ_cooling())
        // ... + time to warm up to T_crossing
        + fridge.tau(fridge.getT_min(), T_crossing, fridge.getQ_warming());
  } else if (doUnload && fridge.isActive()) {
    double temp = fridge.getT_current();
    // T_crossing = temperature where regular and modified will cross
    // the first time
    double T_crossing = fridge.getT_min() + fridge.getT_max() - temp;
    // timespan to this first crossing = time to warm up to T_max ...
    return fridge.tau(temp, fridge.getT_max(), fridge.getQ_warming())
        // ... + time to cool down to T_crossing
        + fridge.tau(fridge.getT_max(), T_crossing, fridge.getQ_cooling());
  }
}
```

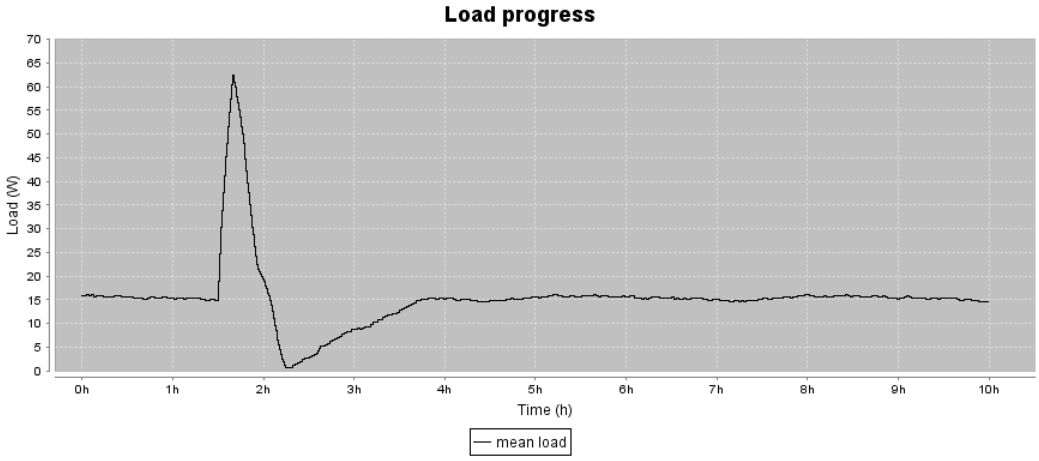
Kurven schneiden. Passend dazu wird dann die Phase ermittelt, in der sich das Gerät zum Zeitpunkt des Schnittpunktes befindet, sowie die dazugehörige Last (q_{max} für Kühlen und q_{min} für Aufwärmen). Dieses Tupel wird dann als Zustandsobjekt state zurückgegeben.

Die Anweisung nextTemp(state) im Ereignis RestoreState ermittelt die „Zieltemperatur“ eines Gerätes mit dem übergebenen Zustand: T_{min} für Kühlen und T_{max} für Aufwärmen. Diese Operation liefert also genau die Temperatur, in der ausgehend vom übergebenen Zustand der nächste Phasenwechsel statt findet.

In Abbildung 5.3 ist das Ergebnis einer Simulation von 5000 Kühlgeräten mit den bekannten Parametern und DSC-*load* Signal zu sehen. Das DSC-Signal wurde wie in Abschnitt 4.1.2 mit $t_{notify} = 90\,min$ und $spread = 10\,min$ konfiguriert. Es wurde der oben beschriebene Controller verwendet, die Kühlgeräte wurden also nach dem Steuersignal durch eine Zustandswiederherstellung desynchronisiert. Es ist auf den ersten Blick zu erkennen, dass die Zustandswiederherstellung offenbar genau den gewünschten Effekt erzielt, es ist im Vergleich zu Abbildung 4.4 keine Oszillation mehr zu erkennen. Allerdings konnte auch keine maximale Lastreduktion auf $q_{min} = 0\,W$ erreicht werden. Insgesamt betrachtet hat sich der Effekt des Steuerungssignales also signifikant verschlechtert, da das primäre Ziel der Lastreduktion nicht mehr



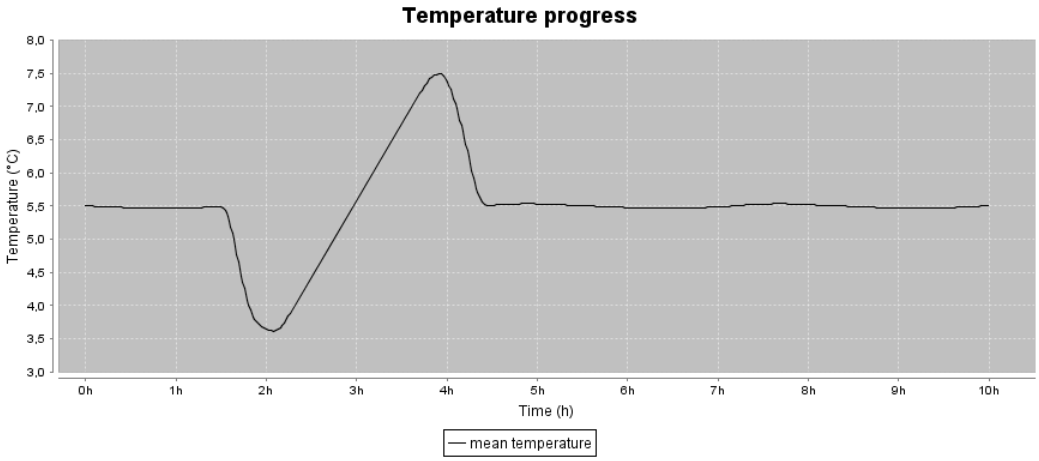
(a)



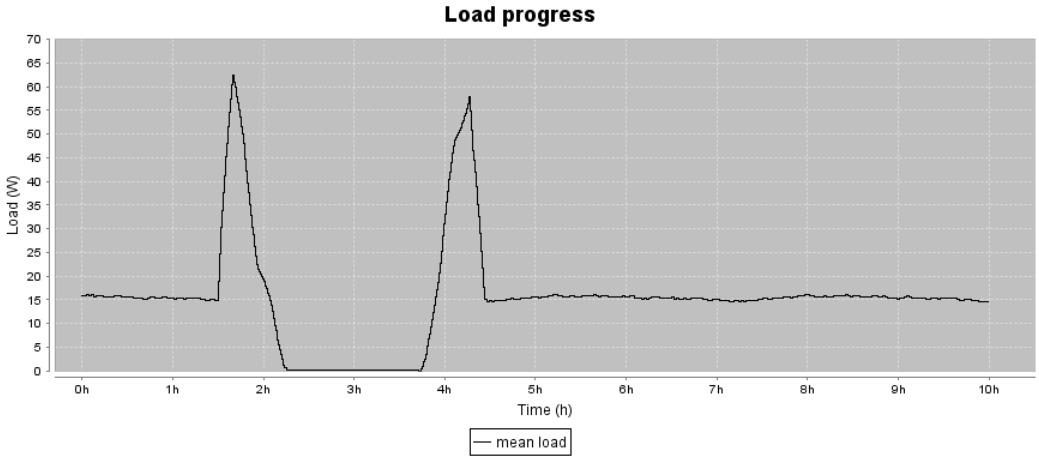
(b)

Abbildung 5.3: Ergebnisse der Zustandswiederherstellung nach DSC-Signal.

genügend erfüllt wird. Um dieses Problem zu beheben, wurde eine zweite Version der Zustandswiederherstellung für das DSC-Signal implementiert. Diese zweite Variante unterscheidet sich von der ersten derart, dass nicht der erste Schnittpunkt der modifizierten und der ursprünglichen Temperaturkurve als Zeitpunkt zur Zustandswiederherstellung verwendet wird, sondern der *zweite*. Es wird also pro Gerät genau ein Zyklus der Dauer $\tau_{cycle} = \tau_{cooling} + \tau_{warming}$ abgewartet, bis der Zustand hergestellt wird, den das Gerät ohne Modifikation gehabt hätte. Diese angepasste Zeitspanne erleichtert zudem die Berechnung des Schnittpunktes und des fiktiven Zustands, da sich der Schnittpunkt eben genau τ_{cycle} Zeiteinheiten in der Zukunft



(a)



(b)

Abbildung 5.4: Ergebnisse der Zustandswiederherstellung nach DSC-Signal, *full-width*-Variante.

befindet, und der Zustand des Gerätes genau dem aktuellen Zustand unmittelbar vor der Modifikation entspricht. Die Ergebnisse einer identischen Simulation mit dieser zweiten Version der Zustandswiederherstellung sind in Abbildung 5.4 dargestellt. Im Vergleich zu den vorherigen Ergebnissen fällt sofort ins Auge, dass einerseits die Lastreduktion genau wie gewünscht erreicht wird, und zudem nach der Phase der reduzierten Last eine vollständige Desynchronisierung statt findet. „Erkauft“ wurde dieses Verhalten durch eine zusätzliche Lastspitze direkt im Anschluss an das Reduktionsintervall. Im folgenden Verlauf dieser Arbeit wird die erste Variante mit der

Bezeichnung *half-width* referenziert, und die zweite mit *full-width*.

Auch für das Steuerungssignal TLR kann eine Zustandswiederherstellung implementiert werden. Hier ist die Anpassung des ursprünglichen TLR-Controllers etwas einfacher, da nicht so stark in die vorhandenen Ereignisse eingegriffen werden muss. Abbildung 5.5 zeigt den erweiterten Controller. Die bekannten Bereiche sind grau hinterlegt, um die neu hinzugekommenen Elemente hervorzuheben. Basierend auf dem Ereignisgraphen in Abbildung 4.8 muss das Ereignis ReduceLoad lediglich um das Sichern des aktuellen Zustandes des Kühlgerätes (in dem Diagramm mit oState, „*original state*“ betitelt) sowie die Einplanung eines zusätzlichen Ereignisses CalculateAction zum Beginn der Reduktionsphase (also mit einer Verzögerung von $\tau_{preload}$ Zeiteinheiten) erweitert werden. Das neue Ereignis CalculateAction berechnet nun auf Basis des Originalzustandes oState sowie der angeforderten Reduktionsdauer τ_{reduce} den fiktiven Zustand, in welchem das Kühlgerät sich nach dem Reduktionsintervall befinden würde, wenn es keine Modifikation gäbe. Dieser Zustand wurde mit rState („*regular state*“) benannt und wird als Parameter an ein weiteres neues Ereignis CalculateState übergeben. Der Zeitpunkt, zu dem CalculateState eingeplant wird, hängt dabei von der Tatsache ab, ob das Kühlgerät in der Lage ist, das angeforderte Reduktionsintervall zu überstehen. Ist diese Bedingung erfüllt ($\tau_{reduce} \leq \tau_{warming}$), so wird das Ereignis direkt nach dem Reduktionsintervall eingeplant. Ist diese Bedingung nicht erfüllt, muss das Kühlgerät bereits vor dem Ende des Reduktionsintervalls wieder beginnen zu kühlen. In diesem Fall wird genau dieser Zeitpunkt, in dem das Gerät seinen Kühlvorgang beginnt, als Einplanzeitpunkt angenommen. Das eingeplante Ereignis CalculateState bestimmt nun auf Basis des ihm übergebenen Zustandes des fiktiven unmodifizierten Gerätes den Zeitpunkt τ_γ, in welchem sich die aktuelle modifizierte sowie die fiktive unmodifizierte Temperaturkurve schneiden würden, und ermittelt den Zustand dState („*desired state*“) des unmodifizierten Gerätes in diesem berechneten Zeitpunkt, und plant darauf basierend die bereits aus der DSC Variante bekannte zweite Modifikation, und damit die eigentliche Zustandswiederherstellung im Ereignis RestoreState ein.

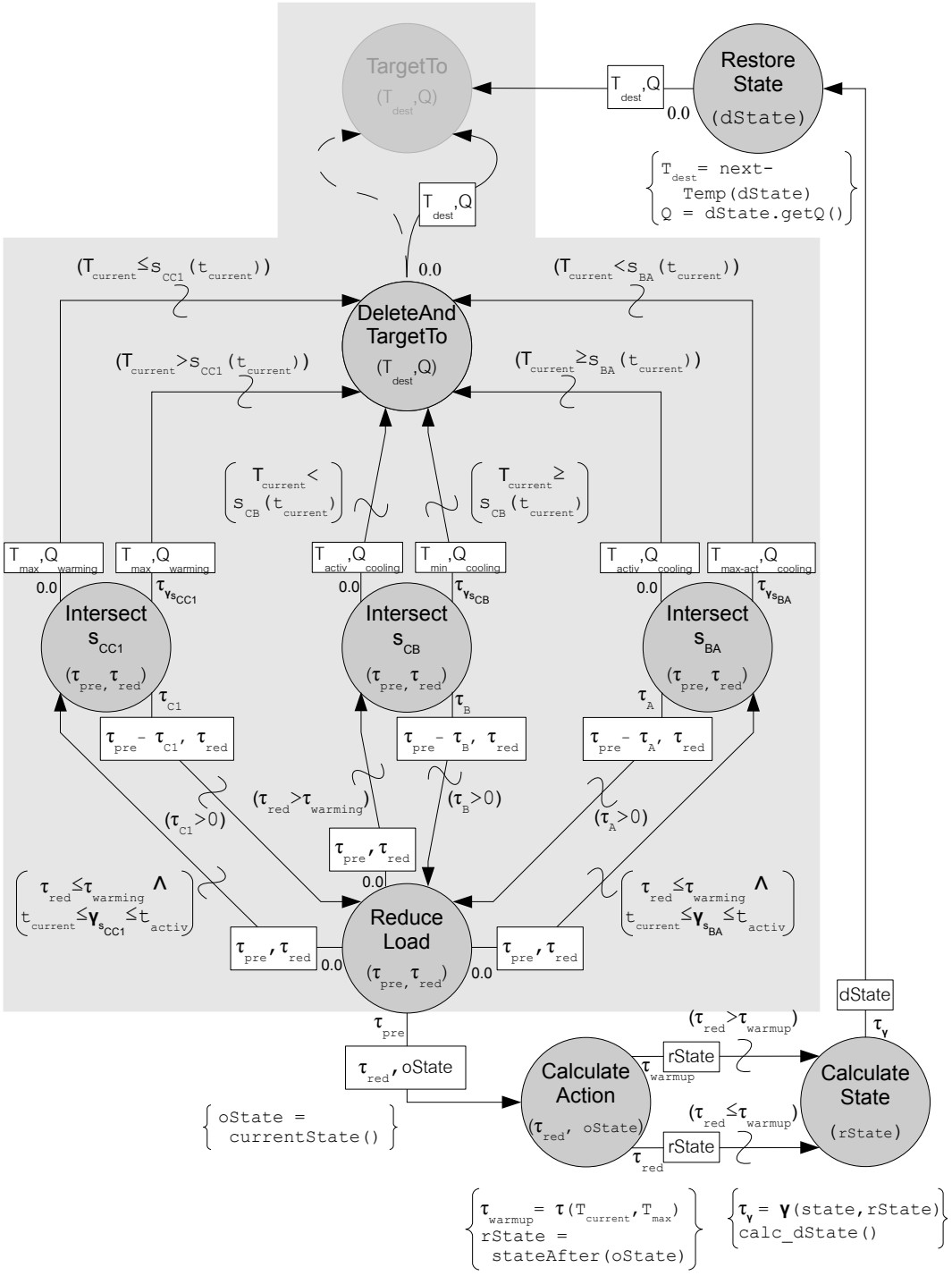
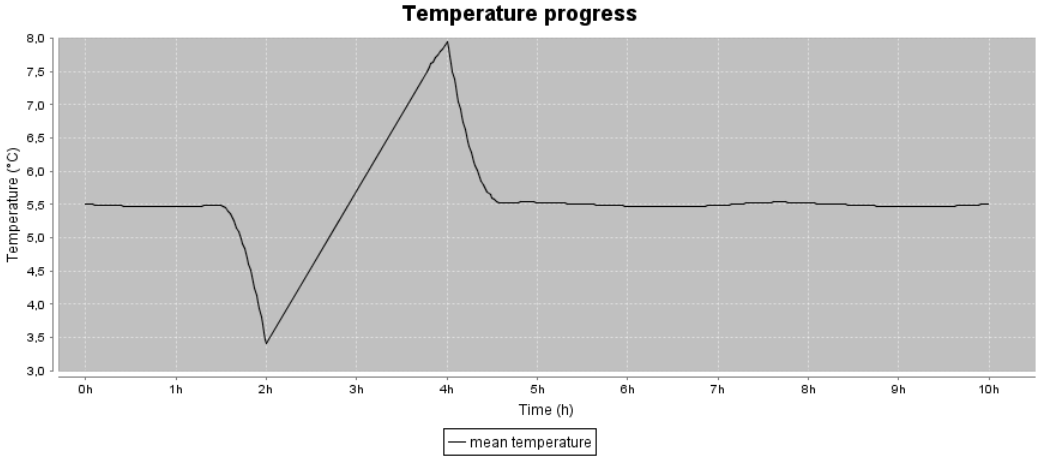
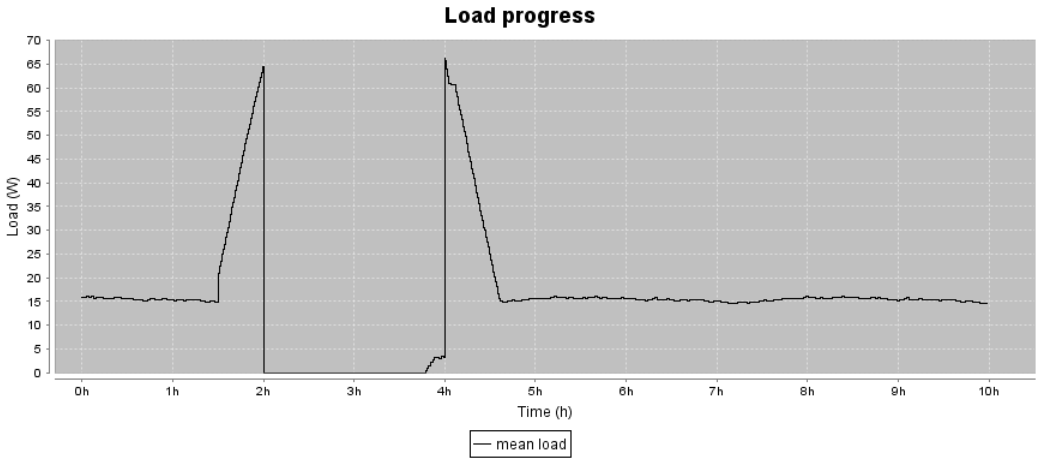


Abbildung 5.5: Linearer TLR-Controller (kompakt) mit Zustandswiederherstellung.

Eine Simulation mit den bekannten Parametern und aktiviertem TLR-Signal mit $t_{notify} = 90\,min$, $\tau_{preload} = 30\,min$ und $\tau_{reduce} = 120\,min$ ergibt die in Abbildung 5.6 dargestellten Ergebnisse.



(a)



(b)

Abbildung 5.6: Ergebnisse der Zustandswiederherstellung nach TLR-Signal.

Im Vergleich zu Abbildung 4.9 ist deutlich zu erkennen, dass die Oszillationen im Anschluss an das Steuerungssignal vollständig gedämpft und die ursprüngliche Zustandsverteilung der Kühlgeräte direkt nach der notwendigen zweiten Lastspitze wieder hergestellt werden konnte.

5.2 Zufallsbasierte Dämpfung

In den verwendeten Simulationseinstellungen sind die Zustände *(Temperatur, Phase, Last)* der Kühlgeräte vor einer Modifikation durch ein Steuerungssignal zufällig verteilt: die Temperatur ist über $[T_{min}, T_{max}]$ gleichverteilt und die Geräte sind über eine Bernoulli-Verteilung mit

$$X \hat{=} \begin{pmatrix} 1 & 0 \\ 0.22 & 0.78 \end{pmatrix}$$

zu etwa 22% in der Phase Kühlen und zu 78% in der Phase Aufwärmen (bzgl. Bernoulli vgl. z. B. [39, Kap. H5.1]). Die Last ist dabei von der Phase abhängig:

$$\text{Kühlen} \rightarrow q_{cooling}\,, \quad \text{Aufwärmen} \rightarrow q_{warming}\,.$$

Um nun nach einem Steuerungssignal die dadurch erfolgte Synchronisation wieder zu dämpfen, sollte es im Gegensatz zur Zustandswiederherstellung ausreichen, eine *ähnliche* Zufallsverteilung herzustellen statt einer exakten Herleitung der regulären Zustände.

Für die Implementierung dieser zufallsbasierten Strategie wurde ein Interface IRandomized angelegt, siehe Listing 5.3. Das Interface schreibt die Implementierung

Listing 5.3: Interface IRandomized

```
1 public interface IRandomized {
2
3   public final static String EV_RANDOMIZE_ACTION = "RandomizeAction";
4
5   public abstract void doRandomizeAction();
6 }
```

des Ereignisses RandomizeAction vor, welches nach erfolgtem Steuerungssignal das folgende Schaltverhalten des Gerätes randomisiert. Für das DSC-Signal wurde der in Abbildung 5.7 dargestellte Controller implementiert. Die bekannten Elemente sind grau hinterlegt. Im Vergleich zu Abbildung 4.3 ist lediglich das vom Interface vorgeschriebene Ereignis RandomizeAction hinzugekommen. Dieses wird nach einer Verzögerung $\tau_{mod} = rnd + \tau_{cooling} + \tau_{warming}$ von den Ereignissen der DSC-Steuerung eingeplant, wobei rnd die bereits aus Abschnitt 4.1.1 bekannte, durch den Parameter spread induzierte Verzögerung beschreibt. Dadurch findet die zweite Modifikation

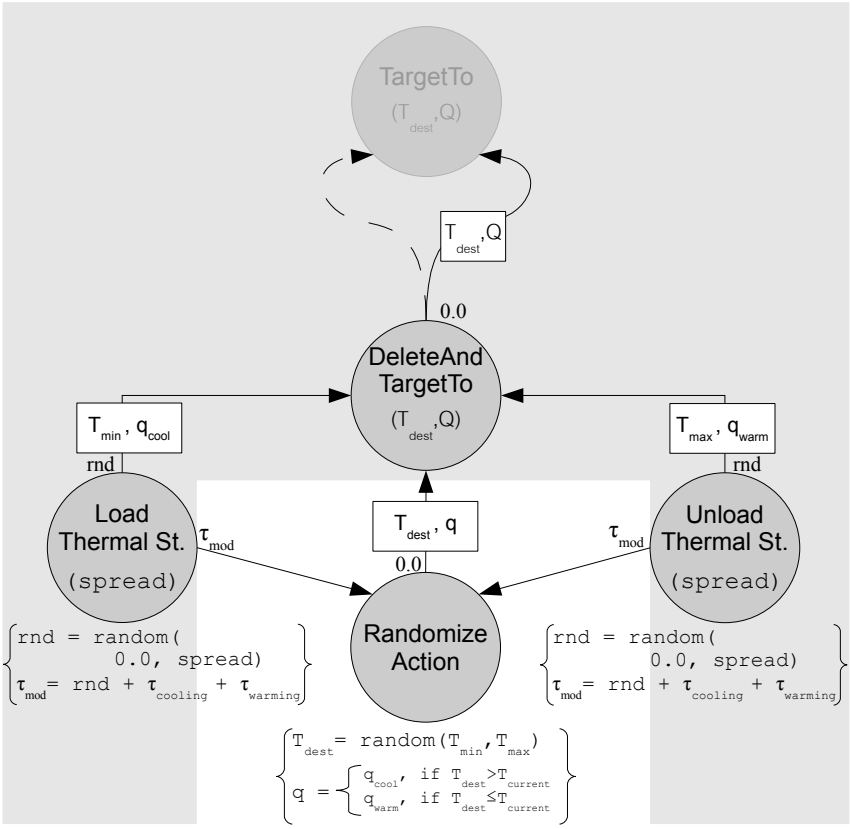
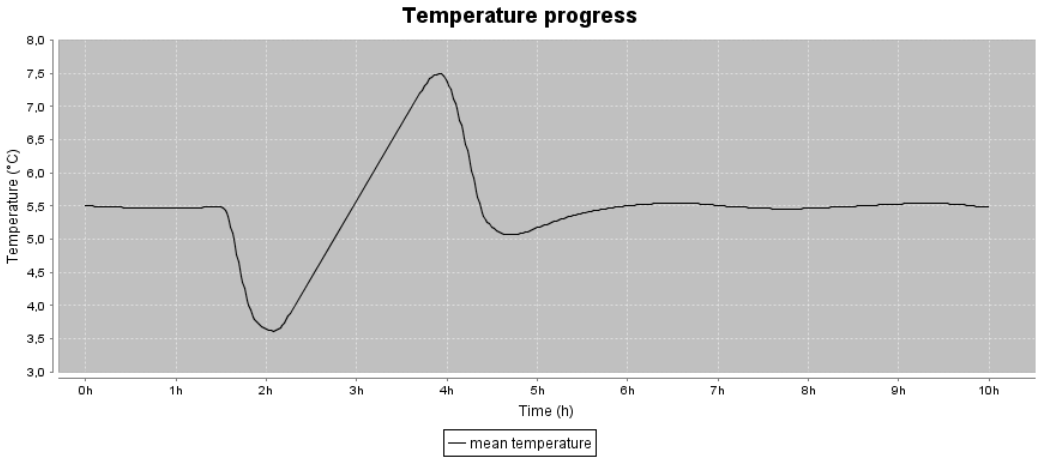
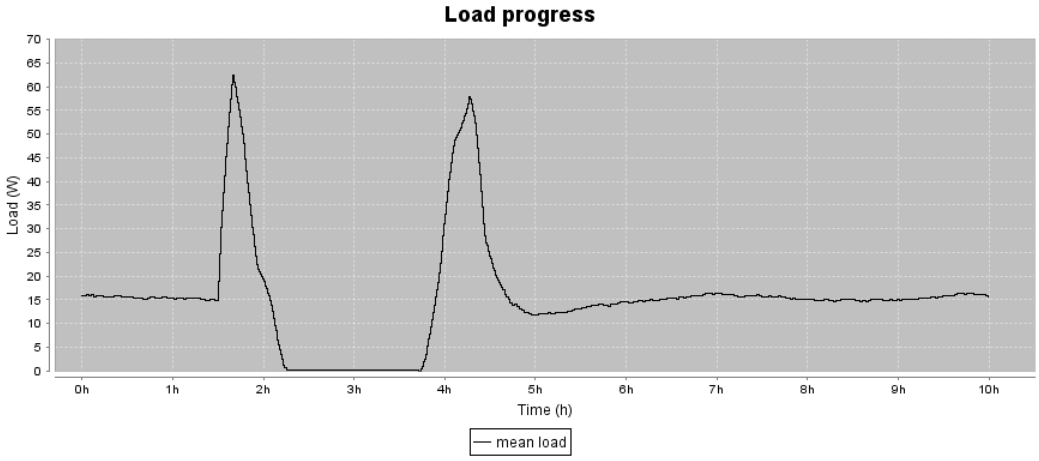


Abbildung 5.7: Linearer DSC-Controller (kompakt) mit zufallsbasierter Dämpfung.

zur Desynchronisierung genau dann statt, wenn das Gerät nach einem DSC-Signal wieder die Temperatur erreicht hat, die das Gerät zum Zeitpunkt der Modifikation durch das Steuersignal hatte. Das neue Ereignis ermittelt nun mittels einer gleichverteilten Zufallszahl einen Temperaturwert aus dem Intervall $[T_{min}, T_{max}]$ und plant das Ereignis DeleteAndTargetTo mit dieser Temperatur und dazu passender Last ein. Durch diese zufallsbasierte Streuung von einer vollständigen Synchronisation in einen gleichverteilten Zustandsraum müsste eine ausreichende Desynchronisation erreicht werden. Abbildung 5.8 zeigt die Ergebnisse einer Simulation mit den bekannten Parametern und der zufallsbasierten Dämpfung nach einem DSC-Signal. Vergleicht man den Kurvenverlauf wiederum mit den Ergebnissen des DSC-Controllers ohne Dämpfungsmechanismus (Abbildung 4.4), so ist auch hier sofort zu erkennen, dass die dauerhafte Synchronisierung der Gerätepopulation erfolgreich verhindert werden konnte.



(a)



(b)

Abbildung 5.8: Ergebnisse der zufallsbasierten Dämpfung nach DSC-Signal.

Für das Steuerungssignal TLR ist diese Methode der Desynchronisation ebenfalls möglich, Abbildung 5.9 zeigt den entsprechenden Ereignisgraphen. Auch hier wurden wieder die bekannten Elemente grau hinterlegt. Es sind genau die gleichen Elemente neu hinzugekommen, die auch in den DSC-Controller integriert wurden. Der Unterschied liegt allein in dem Zeitpunkt, zu dem das Ereignis RandomizeAction eingeplant wird. Bei dem oben eingeführten DSC-Controller mit zufallsbasierter Dämpfung wurde die Verzögerung so gewählt, dass die Ansteuerung einer Zufallstemperatur genau dann erfolgte, wenn das Gerät nach der Modifikation wieder den Zustand erreicht

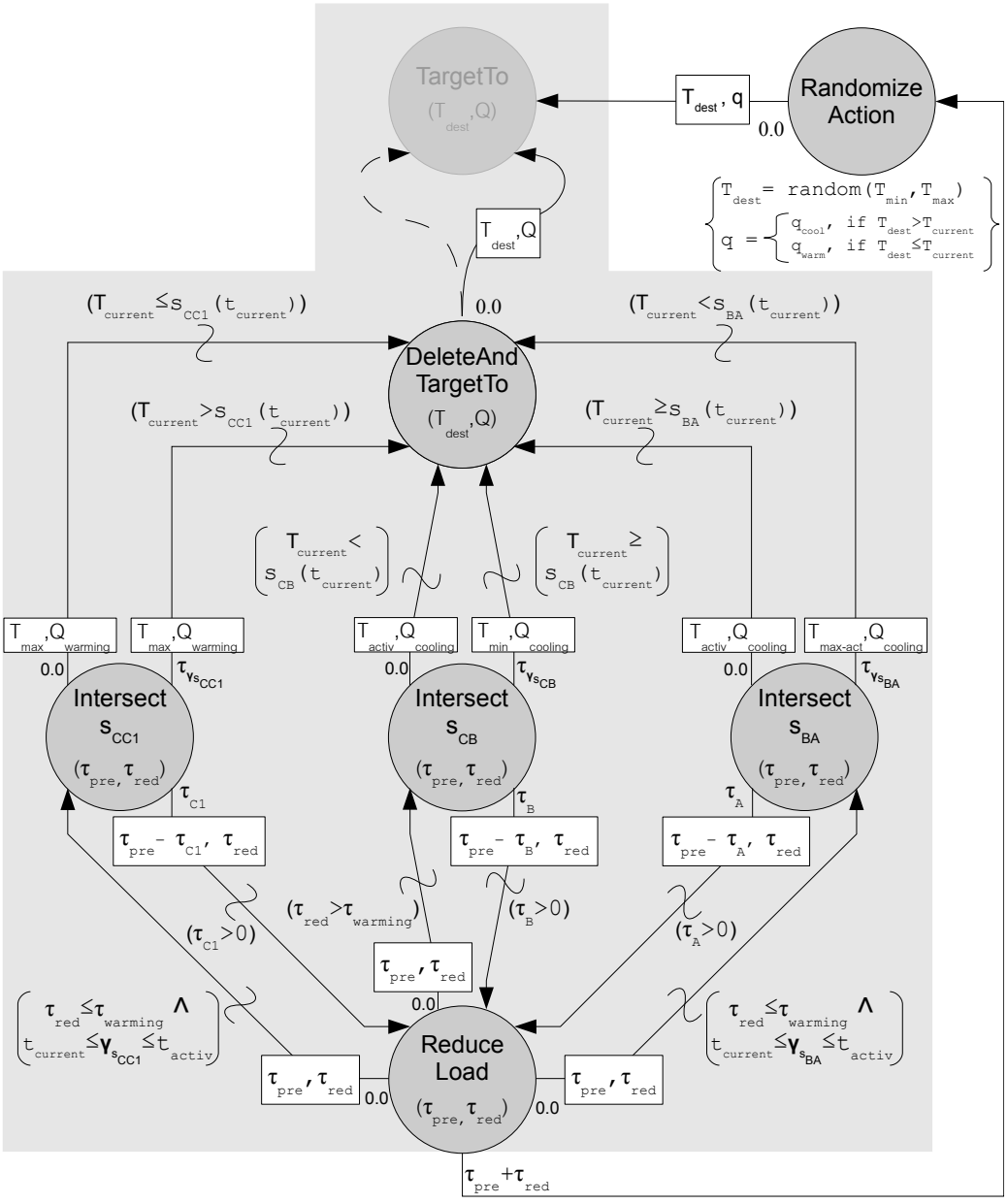
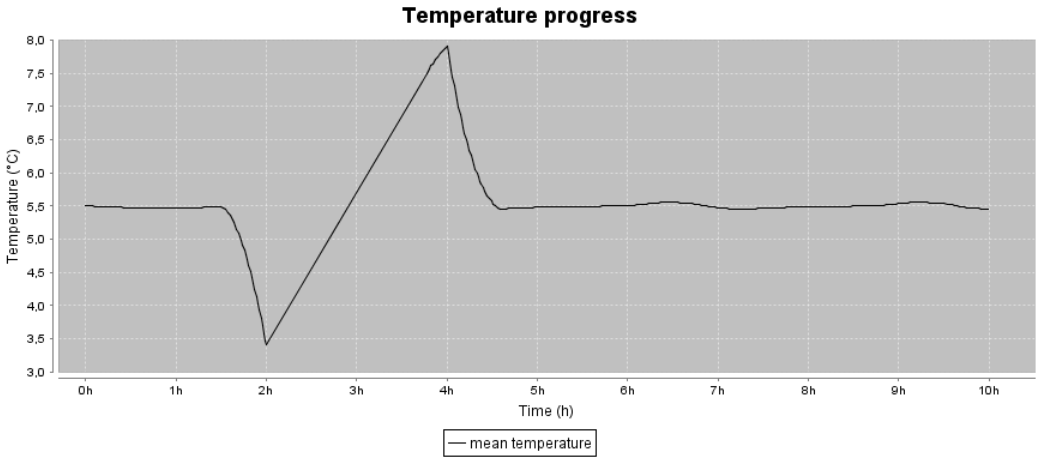
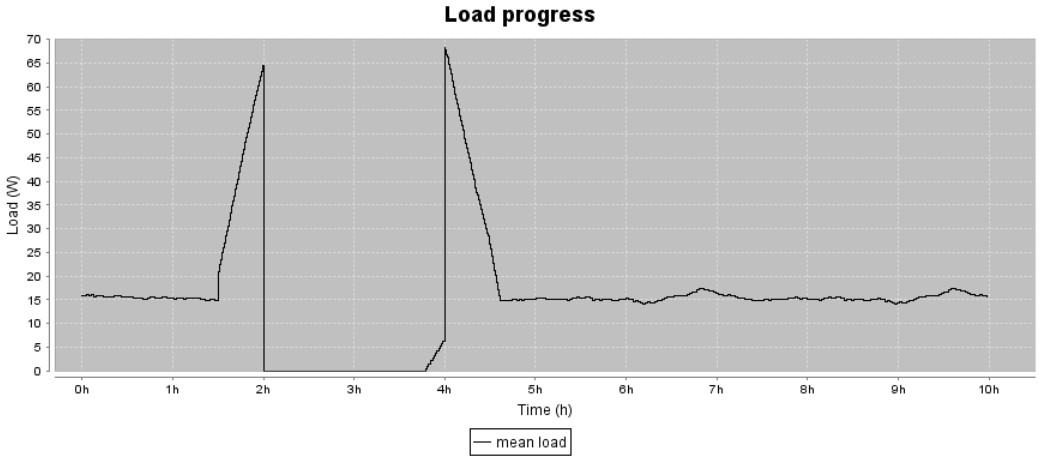


Abbildung 5.9: Linearer TLR-Controller (kompakt) mit zufallsbasierter Dämpfung.



(a)



(b)

Abbildung 5.10: Ergebnisse der zufallsbasierten Dämpfung nach TLR-Signal.

hatte, den es zum Zeitpunkt der Modifikation hatte. Dies ist hier nicht möglich, da während der Vorlaufzeit $\tau_{preload}$ beliebige Zustandsmodifikationen auftreten können. Daher wurde als Zeitpunkt zur Desynchronisation genau das Ende des Reduktionsintervalles gewählt, das Ereignis RandomizeAction wird also mit einer Verzögerung $\tau_{preload} + \tau_{reduce}$ eingeplant. So wird der Ablauf des ursprünglichen TLR-Controllers nicht gestört, und die Desynchronisation kann unmittelbar nach dem Steuerungsintervall erfolgen. In Abbildung 5.10 sind die Ergebnisse einer Simulation mit diesem Controller zu sehen. Wie bei den bisherigen Dämpfungsstrategien kann auch hier eine offensichtliche Desynchronisierung der Geräte festgestellt werden.

5.3 Zusammenfassung

In diesem Kapitel wurden zwei wesentliche Strategien zur Desynchronisierung der Kühlgerätepopulation nach einem erfolgten Steuerungssignal vorgestellt. Die erste Strategie beruht auf der Wiederherstellung von genau der Zustandsverteilung, die unmittelbar vor dem Steuerungssignal herrschte, während die zweite Strategie durch Zufallswerte versucht, eine äquivalent verteilte Zustandsmenge zu erreichen. Erfreulicherweise zeigen beide Strategien für jeweils beide Signaltypen DSC und TLR unter den gegebenen Simulationsparametern sehr gute Desynchronisationsfähigkeiten. Der nächste Schritt besteht darin, die Robustheit dieser Strategien zu untersuchen. In dem folgenden Kapitel wird daher auf abweichende Geräteparameter eingegangen und die erarbeiteten Dämpfungsstrategien unter diesen veränderten Bedingungen getestet.

Kapitel 6

Simulationsparameter

In den bisherigen Simulationsläufen wurden durchgehend die von Stadler et al. untersuchten Standardwerte verwendet. Diese Parameter sind jedoch nur als exemplarisch anzusehen, insbesondere die Geräteparameter der Kühlgeräte sind in der Realität in weit mehr Punkten als nur der Innentemperatur von Gerät zu Gerät unterschiedlich. Dazu zählt etwa die thermische Masse (also der zu kühlende Inhalt des Gerätes), welche einen bedeutenden Einfluss auf die Zeiten $\tau_{cooling}$ und $\tau_{warming}$ ausübt. Auch Parameter wie die Isolierung A in Verbindung mit der Außentemperatur T^O, dem Effizienz-Koeffizient η und der maximalen Leistungsaufnahme Q verändern das Verhalten des Gerätes. Im Folgenden werden diese Parameter daher variiert und die Steuerungssignale DSC und TLR sowie die im vorigen Kapitel erarbeiteten Dämpfungsstrategien auf ihr Verhalten mit diesen abweichenden Parametern untersucht. Insbesondere soll überprüft werden, wie robust die Desynchronisationsmechanismen bei einer Population von Geräten mit gestreuten Parametern sind.

6.1 Grundvoraussetzungen

Bevor jedoch verschiedene Parameter gestreut werden, muss zunächst definiert werden, wie die resultierenden Ergebnisse auszuwerten und zu vergleichen sind. Dazu ist es notwendig, bestimmte Grundvoraussetzungen festzulegen, die die Simulationen unabhängig von den Geräteparametern beeinflussen.
Die Streuung von Parametern in einer Simulation wird mit Hilfe von Pseudozufallszahlen durchgeführt. In Kapitel 4 und 5 wurde bereits die Verwendung von gleichverteilten Zufallszahlen sowie Zufallszahlen nach der Bernoulli-Verteilung erwähnt. In SimKit sind diese und andere Generatoren (auch *variates* genannt) für Zufallszahlen

im Paket simkit.random enthalten. Diese Generatoren können über Parameter weiter konfiguriert werden, so kann etwa für eine Gleichverteilung das Intervall $[min, max]$ der generierten Zahlen festgelegt werden. Weiterhin kann für jeden Generator der zu Grunde liegende Algorithmus zur Erzeugung der Zufallszahlen sowie der *seed*, mit dem der Algorithmus initialisiert wird, manuell gesetzt werden, um etwa bestimmten Anforderungen zu genügen. In der vorliegenden Implementierung wird ein „multiplicative Linear Congruential Generator“ verwendet, der laut A. Buss für jeden beliebigen *seed* eine Periode von 2147483647 aufweist. Die in dieser Arbeit verwendeten *seeds* befinden sich in der Klasse simkit.random.LKSeeds und stammen aus [26].

Für die Auswertung von Simulationsergebnissen gilt nun aber folgendes: Da jede Simulation im Prinzip eine Transformation der Eingabewerte in Ausgabewerte darstellt, kann davon ausgegangen werden, dass bei Zufallszahlen als Eingabewerten auch die Ausgaben zufällig sind. Unter statistischem Gesichtspunkt ergibt sich hier jedoch das Problem, dass die Auswertung und Interpretation von Zufallswerten nur unter bestimmten Bedingungen als korrekt anzusehen ist. Angenommen, es wird eine Simulation des vorgestellten Modells mit beispielsweise 5000 Entitäten ohne Steuerungssignal durchgeführt. Die initalen Temperaturen der Geräte sind gleichverteilt, und zu Beginn der Simulation sind 22% der Geräte in der Phase Kühlen. Die Simulationsdauer beträgt 5 Stunden. Der sich ergebende Verlauf für die mittlere Last aller Geräte bewegt sich um einen aus den Ausgabewerten errechneten Mittelwert von $15.410\,W$. Die Ausgabewerte sind diskreter Natur, jede Änderung mindestens einer simulierten Entität während der Simulation erzeugt einen Datenpunkt *(t,l)*: *(Aktuelle Zeit, Mittlere Last aller Geräte)*. Die Menge dieser Datenpunkte wird als Stichprobe bezeichnet, die einen Ausschnitt aus der Grundgesamtheit darstellt, welche sich ergeben würde, wenn man eine Simulation von unendlicher Dauer und mit unendlich vielen Entitäten durchführen würde. Diese Formulierung deutet bereits auf das Kernproblem hin. Die Grundgesamtheit unterscheidet sich von der Stichprobe einerseits in der Länge der Simulation (also der Anzahl der Datenpunkte), und andererseits in der Menge der simulierten Entitäten, da die Anzahl der Geräte die mittlere Last zu einem beliebigen Zeitpunkt beeinflusst. Diese Parameter werden Freiheitsgrade genannt. Zusätzlich zu diesen beiden lässt sich ein dritter globaler Freiheitsgrad identifizieren: die Zufallszahlen, mit denen die verwendeten Geräteparameter über die Population gestreut werden. Denn unterschiedliche Startbedingungen einer Simulation erzeugen auch abweichende Ergebnisse. Es muss daher zudem untersucht werden, wie oft eine Simulation unter einer bestimmten Konfiguration,

bestehend aus (Geräteparameter, Simulationsdauer, Populationsgröße), mit jeweils verschiedenen Zufallszahlen durchgeführt werden muss, um im Mittel ein einheitliches Ergebnis zu erzielen.
Der errechnete Mittelwert von $15.410\,W$ beschreibt also den mittleren Wert dieser Stichprobe. Die Frage ist nun, inwieweit die Stichprobe die Grundgesamtheit repräsentiert. Diese Abweichung kann durch statistische Verfahren geschätzt werden. Dazu wird an dieser Stelle das Konfidenzintervall verwendet (für die folgenden Erläuterungen vgl. [2, Kap. 11.3 ff] und [40, Kap. 2]). Dieses Intervall $[\bar{x} - h, \bar{x} + h]$ ist so definiert, dass es den errechneten Mittelwert $\bar{x}$ der Stichprobe als Mittelpunkt hat und gerade so breit ist, dass es den wahren Mittelwert μ der Grundgesamtheit mit einer Wahrscheinlichkeit P einschließt. Die Größe h wird als *half-width* bezeichnet. Voraussetzung für die Verwendung von Konfidenzintervallen ist, dass die vorliegende Stichprobe normalverteilt ist. Daher muss zunächst die Verteilung der Stichprobe untersucht werden. Dazu werden die Werte der Stichprobe nach ihrem Rang sortiert, Duplikate entfernt und dann gegen eine ebenso behandelte synthetisch erzeugte Normalverteilung mit der gleichen Länge und gleichem Mittelwert sowie Standardabweichung in einem Diagramm aufgetragen (vlg. z. B. [24, Abs. 7.4]). Dieses Diagramm wird als Quantile-Quantile-Diagramm bezeichnet und ist in Abbildung 6.1 dargestellt. Die durchgezogene Gerade zeigt den Verlauf einer optimal auf

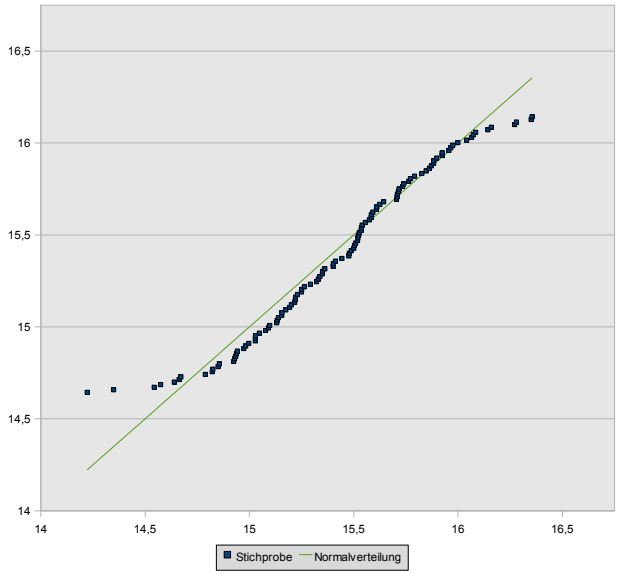


Abbildung 6.1: Quantile-Quantile-Diagramm zur Prüfung auf Normalverteilung.

die Referenzverteilung passende Stichprobe. Die vorliegende Stichprobe nähert sich dieser Geraden im mittleren Bereich sehr gut an, nur an den Enden zeigt sie einen etwas abweichenden, S-förmigen Verlauf. Diese Abweichung deutet auf eine supergaussförmige, bzw. leptokurtische Verteilung hin, welche der Normalverteilung sehr ähnlich ist, aber genau wie die t-Verteilung eine leicht andere Wölbung aufweist. Es sei daher an dieser Stelle darauf hingewiesen, dass die folgenden Berechnungen unter nicht optimalen Bedingungen stattfinden, jedoch für den Kontext dieser Arbeit ausreichend sein sollten.

Das Konfidenzintervall mit einem Vertrauensniveau $P = (1 - \alpha) \cdot 100\%$ für den Mittelwert $\bar{x}$ einer Stichprobe der Größe N kann nun durch folgende Formel ermittelt werden:

$$h = t_{N-1,\frac{\alpha}{2}} \frac{\sigma}{\sqrt{N}} \tag{6.1}$$

Dabei bezeichnet $t_{N-1,\frac{\alpha}{2}}$ das $\frac{\alpha}{2}$-Quantil der Student'schen t-Verteilung mit $N - 1$ Freiheitsgraden, und σ die Standardabweichung der Stichprobe. Der errechnete Wert gibt also Aufschluss darüber, wie weit der wahre Mittelwert μ vom errechneten Mittelwert $\bar{x}$ abweicht. Üblicherweise wird für das Vertrauensniveau ein $\alpha = 0.05$ verwendet, sodass das ermittelte Konfidenzintervall mit einer Wahrscheinlichkeit von 95% korrekt ist. Für die vorliegende Fragestellung ist diese Berechnung jedoch noch nicht sonderlich hilfreich, da nicht die Genauigkeit einer bestimmten Stichprobe untersucht werden soll, sondern die umgekehrte Fragestellung, wie die Stichprobe anzupassen ist, um eine bestimmte Genauigkeit zu erhalten. Eine Umformung der Berechnung ergibt folgende Gleichung:

$$\tilde{N} = \left(z_{\frac{\alpha}{2}} \frac{\sigma}{\tilde{h}} \right)^2 \tag{6.2}$$

Diese Formel liefert die geschätzte notwendige Größe $\tilde{N}$ der Stichprobe, um ein Konfidenzintervall der Breite $2\tilde{h}$ mit Wahrscheinlichkeit α um den Mittelwert $\bar{x}$ einer initialen Stichprobe der Größe N zu erhalten. Da der zuvor verwendete Faktor $t_{N-1,\frac{\alpha}{2}}$ seinerseits von N abhängt, wurde er in dieser Berechnung durch den entsprechenden Wert $z_{\frac{\alpha}{2}}$ der Standardnormalverteilung ersetzt, welche bei den verwendeten Stichprobengrößen eine ausreichende Annäherung an die t-Verteilung liefert. Mit Hilfe dieser Formeln können nun die Freiheitsgrade „Simulationsdauer“, „Populationsgröße“ und „Zufallszahlenvariation“ untersucht werden. Die Faktoren t und z

sind dabei den Tabellen in einschlägiger Statistikliteratur zu entnehmen oder durch ihre Berechnungsvorschriften (etwa über eine Statistiksoftware) zu ermitteln.

6.1.1 Simulationsdauer

Zunächst soll die Länge der Simulation betrachtet werden. Prinzipiell ist dies der unwichtigste Faktor in der vorliegenden Aufgabenstellung, da die notwendige Länge der Simulation durch die Aktionsphase der zu untersuchenden Steuerungssignale bereits fest definiert ist. Eine Korrektur der mittleren Werte einer zufallsbasierten Streuung durch eine längere Simulation ist somit hier wenig sinnvoll, da das zu untersuchende Zeitintervall von einer längeren Simulationsdauer nicht verändert wird. Es wird dennoch das oben vorgestellte Verfahren angewendet, um seine Funktionsweise zu erläutern. Zunächst wird das initiale Konfidenzintervall mit *half-width* h_0 ermittelt, indem die Datenpunkte *(t,l)* der oben vorgestellte Simulation ausgewertet werden. Es ergibt sich die in Tabelle 6.1 dargestellte Stichprobe. Mittels Formel 6.1

Größe der Stichprobe	5 Stunden
Minimum	14.378 W
Maximum	16.506 W
Mittel	15.410 W
Varianz	0.274
Standardabweichung	0.523 W

Tabelle 6.1: Initiale Stichprobe (Dauer).

ergibt sich für diese Stichprobe mit $\alpha = 0.05$:

$$h_0 = t_{4,0.025} \cdot \frac{0.523\,W}{\sqrt{5}} = 2.78 \cdot \frac{0.523\,W}{\sqrt{5}} \approx 0.651\,W$$

Das bedeutet, dass der wahre Mittelwert mit einer Wahrscheinlichkeit von 95% im Intervall $[14.759\,W, 16.061\,W]$ liegt. Möchte man dieses Intervall nun beispielsweise auf $\tilde{h} = \frac{h_0}{2} = 0.326\,W$ eingrenzen, um die Genauigkeit der Simulation zu erhöhen, so ergibt sich nach Formel 6.2 die folgende Rechnung:

$$\tilde{N} = \left(z_{0.025} \cdot \frac{0.523\,W}{0.326\,W} \right)^2 = \left(1.96 \cdot \frac{0.523\,W}{0.326\,W} \right)^2 \approx 3.153^2 \approx 9,942$$

Es wäre also eine Simulation mit einer Dauer von mindestens 9,942 Stunden notwendig, damit der wahre Mittelwert μ mit einer Wahrscheinlichkeit von 95% maximal um $\tilde{h} = 0.326\,W$ vom errechneten Mittelwert $\bar{x}$ der Stichprobe abweicht.

Bei dieser Rechnung ist die Größe $\tilde{h} = \frac{h_0}{2}$ frei gewählt und eher exemplarisch, die anzustrebende Genauigkeit sollte in Abhängigkeit der von dieser Größe beeinflussten Werte in einem sinnvollen Rahmen festgelegt werden.
Für die durchzuführenden Simulationsläufe wird weiterhin wie gehabt eine Dauer von 10 Stunden angesetzt, da die Anpassung der Simulationslänge wie eingangs erwähnt keine höhere Genauigkeit bei der Analyse der Steuerungssignale bietet und dieser Abschnitt allein der Demonstration der Formeln dient.

6.1.2 Populationsgröße

Ein im Gegensatz zum vorherigen Abschnitt bedeutsamerer Freiheitsgrad ist die Populationsgröße, also im vorliegenden Fall die Menge der simulierten Kühlgeräte. Da einige Geräteparameter der Entitäten zu Beginn der Simulation mittels Zufallszahlen gestreut werden, wird die angestrebte statistische Verteilung dieser Parameter umso sichtbarer, je mehr Entitäten beteiligt sind. Während der bisher vorgenommenen Experimente im Laufe dieser Arbeit wurde stets eine Populationsgröße von 5000 Geräten verwendet, wie es von Stadler et al. vorgeschlagen wurde. Dieser Wert soll nun auf seine Auswirkungen auf die Genauigkeit der Simulation analysiert und bei Bedarf angepasst werden.
Das initiale Konfidenzintervall mit *half-width* h_0 wird somit durch eine Stichprobe errechnet, welche sich aus einer Simulation mit den oben verwendeten Werten mit einer Populationsgröße von $N = 5000$ ergibt. Tabelle 6.2 zeigt die resultierenden Werte. Mittels Formel 6.1 ergibt sich für diese Stichprobe mit $\alpha = 0.05$:

$$h_0 = t_{4999,0.025} \cdot \frac{0.494\,W}{\sqrt{5000}} = 1.645 \cdot \frac{0.494\,W}{\sqrt{5000}} \approx 0.012\,W$$

Der wahre Mittelwert der Grundgesamtheit unter den Voraussetzungen der Konfiguration dieser Simulation liegt also mit einer Wahrscheinlichkeit von 95% im Intervall $[15.436\,W, 15.459\,W]$. Die geschätzte maximale Abweichung h_0 zum wahren Mittelwert beträgt demnach 0.074% des errechneten Mittelwertes. Diese Größenordnung ist absolut akzeptabel, die Genauigkeit der Simulation ist mit der vorgeschlagenen Populationsgröße von 5000 Geräten also vollkommen ausreichend.

Größe der Stichprobe	5000 Geräte
Minimum	14.378 W
Maximum	16.506 W
Mittel	15.448 W
Varianz	0.244
Standardabweichung	0.494 W

Tabelle 6.2: Initiale Stichprobe (Populationsgröße).

Um dieses Ergebnis zu verifizieren wird die Rechnung für eine höhere Aussagewahrscheinlichkeit von 99% wiederholt:

$$h_0 = t_{4999,0.005} \cdot \frac{0.494\,W}{\sqrt{5000}} = 2.327 \cdot \frac{0.494\,W}{\sqrt{5000}} \approx 0.016\,W$$

Das Konfidenzintervall weitet sich bei dieser hohen Aussagewahrscheinlichkeit also auf $[15.432\,W, 15.464\,W]$ aus, das geschätzte h_0 beträgt hier 0.105% des errechneten Mittelwertes. Es zeigt sich also, dass eine Population von 5000 Geräten mit sehr hoher Wahrscheinlichkeit für eine ausreichend genaue Simulation groß genug ist.

6.1.3 Zufallszahlenvariation

Nachdem die notwendige Populationsgröße geschätzt wurde, soll nun überprüft werden, welche Auswirkung die Verwendung von unterschiedlichen Zufallszahlenmengen auf das Simulationsergebnis hat. Dazu wird obige Simulation mit unveränderten Parametern in 5 Durchläufen mit gleicher Verteilung der Zufallszahlen (also eine Gleichverteilung der Starttemperaturen, und eine Bernoulli-Verteilung der Startaktivität), aber mit variierenden Ausprägungen der einzelnen Zufallszahlen durchgeführt. Diese Variation wird durch unterschiedliche *seeds* für die Generatoren realisiert. Für jeden Durchlauf wird die mittlere Last über den gesamten Simulationszeitraum als statistischer Lageparameter gesichert. Diese fünf Werte bilden dann die Stichprobe, über welche das initiale Konfidenzintervall gebildet wird. Tabelle 6.3 zeigt die resultierenden Werte. Nach Formel 6.1 ergibt sich wiederum für $\alpha = 0.05$ das folgende h_0:

$$h_0 = t_{4,0.025} \cdot \frac{0.025\,W}{\sqrt{5}} = 2.132 \cdot \frac{0.025\,W}{\sqrt{5}} \approx 0.024\,W$$

Das initiale Konfidenzintervall ist also $[15.398\,W, 15.446\,W]$, die geschätzte maximale Abweichung h_0 beträgt 0.155% des Mittelwertes der Stichprobe. Dieser Wert

Simulationslauf	*Mittelwert der Last*
1	15.448 W
2	15.427 W
3	15.382 W
4	15.437 W
5	15.417 W
Statistische Auswertung der Stichprobe	
Minimum	15.382 W
Maximum	15.448 W
Mittel	15.422 W
Varianz	0.001
Standardabweichung	0.025 W

Tabelle 6.3: Initiale Stichprobe (Zufallszahlenvariation).

ist wie auch die Schätzung der Populationsgröße im vorhergehenden Abschnitt im akzeptablen Bereich und muss nicht weiter verfeinert werden. Zur Bewertung dieser Aussage wurde eine weitere Schätzung mit 20 Simulationsdurchläufen und Zufallszahlenvariation vorgenommen, die Analyse dieser Stichprobe ergab eine geschätzte maximale Abweichung von 0.057%. Durch eine Vervierfachung der Simulationsläufe konnte eine gute Verdopplung der Genauigkeit erreicht werden. Es ist also möglich, durch eine Erhöhung der Simulationsläufe noch deutliche Verbesserungen der Genauigkeit zu erlangen, dies ist jedoch bei den vorliegenden Größenordnungen nicht unbedingt notwendig.
Da auch dieses Ergebnis dem Simulationsanalysten sehr entgegen kommt und wünschenswert ist, wird zur Sicherheit wie im vorigen Abschnitt eine Bestätigungsrechnung mit einer Aussagewahrscheinlichkeit von 99% durchgeführt:

$$h_0 = t_{4,0.005} \cdot \frac{0.025\,W}{\sqrt{5}} = 3.746 \cdot \frac{0.025\,W}{\sqrt{5}} \approx 0.042\,W$$

Die Rechnung ergibt ein erweitertes Konfidenzintervall von $[15.378\,W, 15.464\,W]$, die *half-width* beträgt 0.273% des errechneten Mittelwertes. Dieser Wert ist knapp doppelt so groß wie der Wert der 95%-Rechnung, bewegt sich jedoch ebenfalls weit

unterhalb der Grenze von 1% Abweichung und bestätigt somit die obige Rechnung. Es wird daher im Folgenden mit jeweils fünf Durchläufen pro Simulation gearbeitet.

6.1.4 Kombination der Schätzungen

Im Abschnitt 6.1.2 wurde die notwendige Populationsgröße für aussagekräftige Simulationen geschätzt. Diese Schätzung basierte jedoch auf einer bestimmten Menge von Zufallszahlen, es könnte also sein, dass die dort gewonnenen Werte nicht repräsentativ sind. Zur Validierung wird die gleiche Analyse für vier weitere Simulationen mit anderen Zufallszahlen durchgeführt. Es ergeben sich die in Tabelle 6.4 dargestellten Ergebnisse. Die Werte zeigen, dass sich die Schätzungen bei variierenden Zufallszahlenmengen kaum verändern, die Spannweite liegt bei etwa 0.013 Prozentpunkten. Dieses Ergebnis validiert somit die Aussage, dass mit einer Populationsgröße von 5000 Geräten und einer fünfmaligen Wiederholung der Simulation mit unterschiedlichen Zufallszahlen Simulationsergebnisse erzielt werden, die auf den Mittelwert bezogen mit einer Aussagewahrscheinlichkeit von 99% weit weniger als 1% Abweichung zur theoretischen Grundgesamtheit aufweisen.

Simulationslauf	*Geschätzte prozentuale Abweichung*
1	0.074%
2	0.077%
3	0.087%
4	0.085%
5	0.078%
Statistische Auswertung der Stichprobe	
Minimum	0.074%
Maximum	0.087%
Mittel	0.080%
Varianz	0.00003
Standardabweichung	0.006%

Tabelle 6.4: Validierung der Schätzung der Populationsgröße.

Eine weitere Möglichkeit, die Simulationsparameter auf ihre Genauigkeit hin zu überprüfen, besteht in der Analyse der Varianzen der Stichproben. Hier bietet sich der sogenannte F-Test an (siehe [24, Abs. 19.5]), mit welchem sich signifikante Unterschiede zwischen Varianzen aufdecken lassen. Für zwei Stichproben X_1 und X_2 wird dazu mittels der folgenden Teststatistik der zugehörige F-Wert gebildet:

$$F^{(n_1-1,n_2-1)} = \frac{\sigma_1^2}{\sigma_2^2} \geq 1 \qquad \text{für} \quad \sigma_1^2 \geq \sigma_2^2 \,. \tag{6.3}$$

Dabei bezeichnen n_1 und n_2 den Umfang und σ_1^2 sowie σ_2^2 die Varianz der jeweiligen Stichprobe X_1 und X_2. Es muss darauf geachtet werden, dass X_1 und X_2 nach der Größe ihrer Varianz sortiert sind, sodass der Quotient einen Wert ≥ 1 liefert. Der errechnete F-Wert lässt sich dann mittels der F-Verteilung[1] in eine Irrtumswahrscheinlichkeit $\hat{\alpha}$ übersetzen. Ist diese ermittelte Wahrscheinlichkeit größer als das angestrebte Testniveau α, so muss davon ausgegangen werden, dass sich die beiden Stichproben signifikant in ihrer Varianz unterscheiden. Für die oben durchgeführten Simulationen ergeben sich die in Tabelle 6.5 aufgelisteten F-Werte.

Simulationsläufe	*F-Wert*	$\hat{\alpha}$
1, 2	$F^{(36124,36124)} = \frac{0.258}{0.244} = 1.057$	0.0
1, 3	$F^{(35942,36124)} = \frac{0.330}{0.244} = 1.353$	0.0
1, 4	$F^{(36067,36124)} = \frac{0.318}{0.244} = 1.303$	0.0
1, 5	$F^{(36019,36124)} = \frac{0.265}{0.244} = 1.086$	0.0
2, 3	$F^{(35942,36124)} = \frac{0.330}{0.258} = 1.279$	0.0
2, 4	$F^{(36067,36124)} = \frac{0.318}{0.258} = 1.233$	0.0
2, 5	$F^{(36019,36124)} = \frac{0.265}{0.258} = 1.027$	0.0057
3, 4	$F^{(35942,36067)} = \frac{0.330}{0.318} = 1.038$	0.0002
3, 5	$F^{(35942,36019)} = \frac{0.330}{0.265} = 1.245$	0.0
4, 5	$F^{(36067,36019)} = \frac{0.318}{0.265} = 1.2$	0.0

Tabelle 6.5: Untersuchung der Varianzen mittels F-Test.

[1] siehe etwa http://www.ggediga.de/w3lib/fvert.htm, zuletzt besucht am 24. Oktober 2008

Die Tabelle zeigt, dass sich die Varianzen der einzelnen Simulationsläufe nur sehr geringfügig unterscheiden (Werte auf vier Nachkommastellen gerundet). Der größte $\hat{\alpha}$-Wert liegt bei 0.0057, damit lässt sich etwa für ein angestrebtes Testniveau von $\alpha = 0.01$ in keiner Kombination der durchgeführten Simulationen eine signifikante Abweichung der Varianz festellen. Die ausgewählten Simulationsparameter sind also auch nach diesem Test mit einer Wahrscheinlichkeit von 99% für ausreichend genaue Aussagen geeignet.

6.2 Geräteparameter

Im vorigen Abschnitt wurden die globalen Simulationsparameter festgelegt, nun soll untersucht werden, wie sich eine Streuung der individuellen Geräteparameter auf die Ergebnisse der Simulation auswirkt. Als erstes wird die Verteilung der Start-

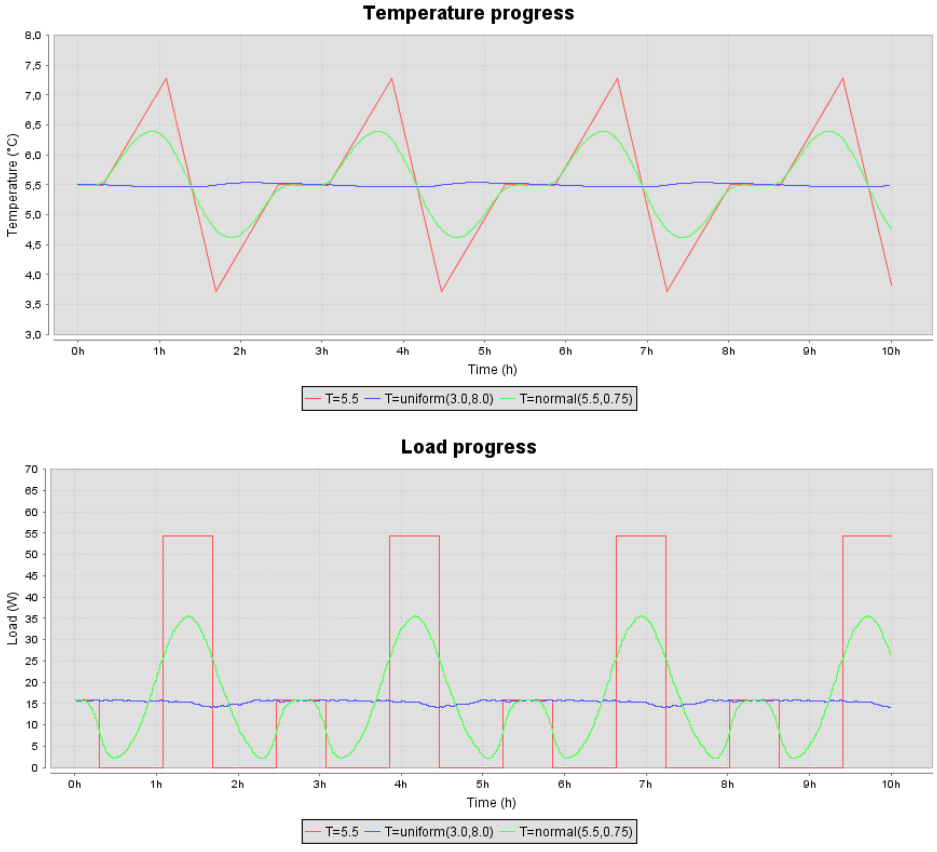


Abbildung 6.2: Verschiedene Streuungen der Starttemperatur.

temperaturen bei einer Simulation ohne Steuerungssignal betrachtet. Abbildung 6.2 zeigt den Kurvenverlauf für Konfigurationen mit festem $T_{start} = 5.5°C$, gleichverteilt im Intervall $[3\,°C, 8\,°C]$ sowie normalverteilt mit $\mu = 5.5\,°C$ und $\sigma = 0.75$. Die übrigen Parameter sind auf ihre festen Standardwerte eingestellt. Es ist deutlich zu sehen, dass sich nur mit gleichverteilten Starttemperaturen ein homogener Kurvenverlauf ergibt. Es stellt sich nun die Frage, ob der gleichmäßige Verlauf allein durch diesen Parameter bestimmt wird, oder ob ein ähnliches Bild auch mit anderen Parametern erreicht werden kann. In Abbildung 6.3 sind daher einige Experimente mit variierenden Kombinationen von Streuungen der Starttemperatur T_{start} sowie der thermischen Masse m_c dargestellt. Bei T_{start} verhält es sich wie oben, und m_c wurde

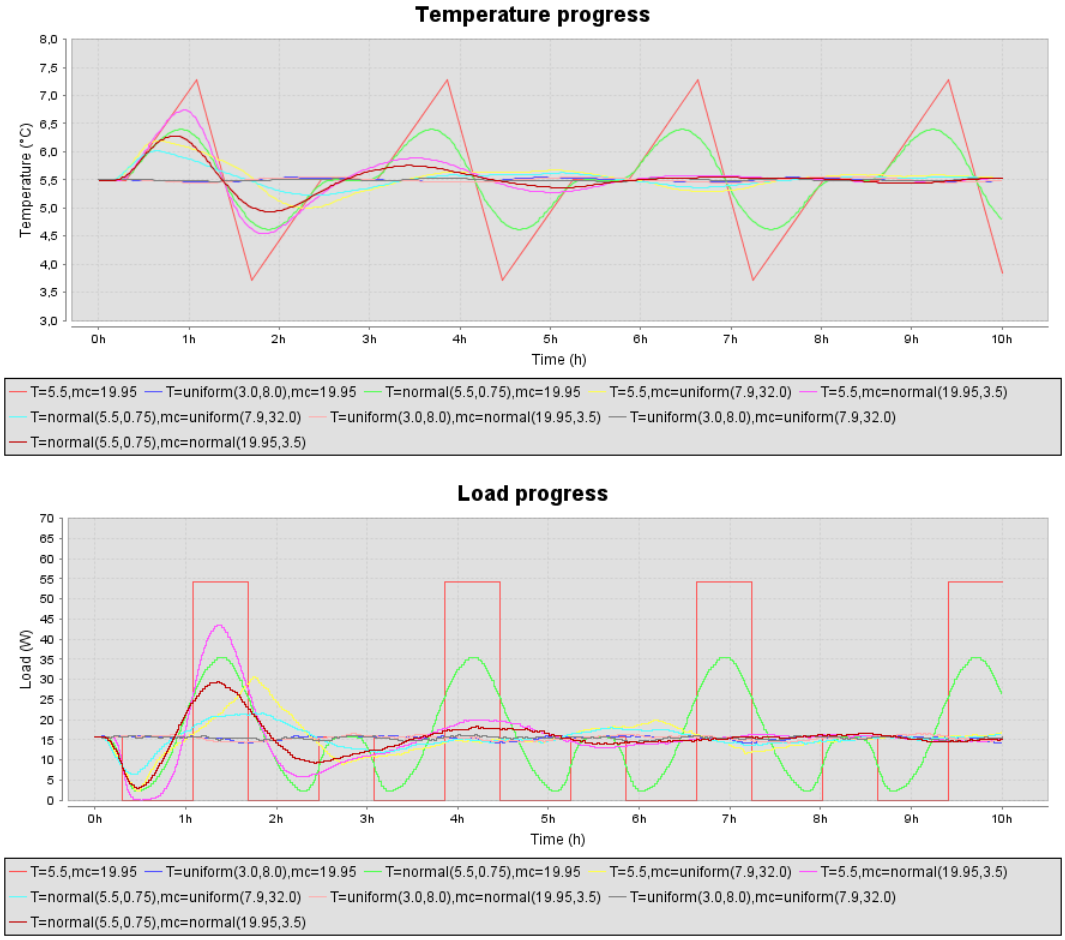
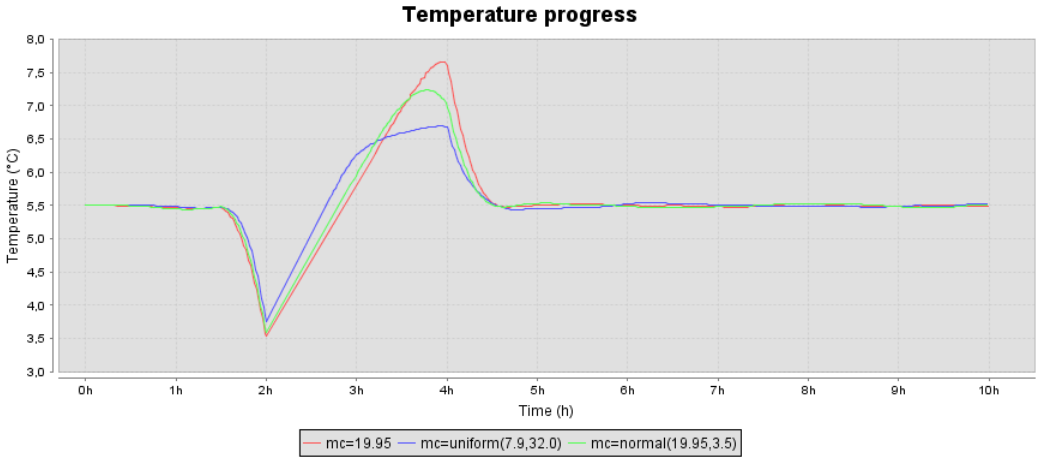


Abbildung 6.3: Verschiedene Streuungen der Starttemperatur und der thermischen Masse.

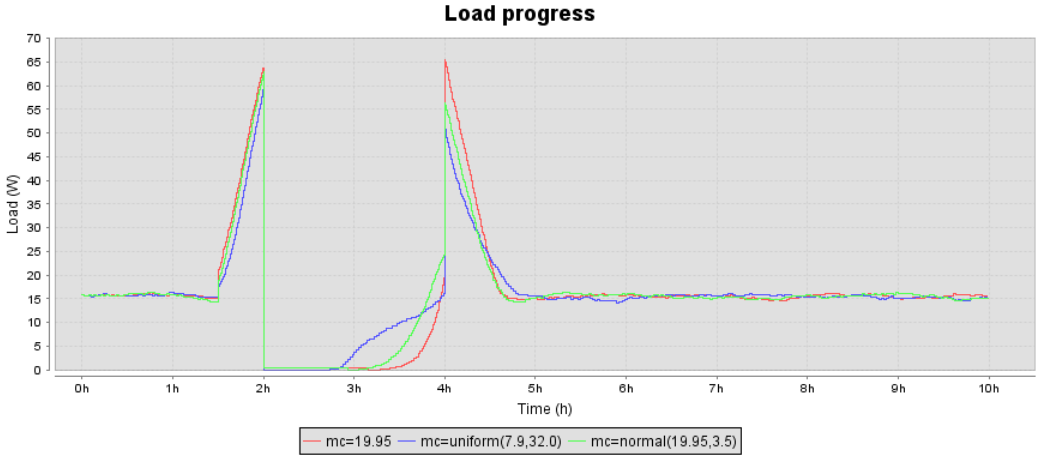
ebenfalls in drei Varianten verwendet: als fester Wert $m_c = 19.95\,\frac{kW}{°C}$, gleichverteilt im Intervall $[7.9\,\frac{kW}{°C}, 32.0\,\frac{kW}{°C}]$, und abschließend normalverteilt mit $\mu = 19.95\,\frac{kW}{°C}$ und $\sigma = 3.5$. Die Auswertung zeigt mehrere Dinge: zunächst ist zu erkennen, dass drei Varianten (dunkelblau, grau und rosa) einen sehr homogenen Verlauf bewirken, dies sind genau die Kombinationen, in denen die Starttemperatur gleichverteilt war. Es spielt offenbar also keine Rolle, wie die thermische Masse ausgeprägt ist, solang die Starttemperatur gleichmäßig über die Population verteilt ist. Für die anderen Konfigurationen können prinzipiell zwei Fälle unterschieden werden. Der erste beinhaltet die hellrote und die grüne Kurve, die auch schon in Abbildung 6.2 enthalten waren. Diese Kombinationen zeichnen sich dadurch aus, dass entweder gar kein Parameter gestreut wurde (hellrot), oder dass nur die Starttemperatur normalverteilt ist, während die anderen Parameter (hier insbesondere die thermische Masse) auf einen Festwert eingestellt sind. Diese beiden Kurven behalten ihren charakteristischen Verlauf über die Zeit bei und zeigen somit eine deutliche Schwingung, deren Frequenz die Zeitspanne $\tau_{cycle} = \tau_{cooling} + \tau_{warming}$ ist. Die wichtigste Erkenntnis steckt jedoch im zweiten Fall, der für alle übrigen Kurven gilt. Hier ist zu sehen, dass die Kurven sich nach einer mehr oder weniger langen Einschwingphase einem homogenen Verlauf um den Mittelwert herum annähern.
Diese drei Betrachtungen lassen nun folgenden Schluss erahnen: eine gleichmäßige Verteilung der Zustände der einzelnen Geräte lässt sich entweder dadurch erreichen, dass die Innentemperaturen schon von Beginn der Simulation an über das erlaubte Intervall gleichverteilt sind, oder durch eine Variation der Zeitspannen $\tau_{cooling}$ und $\tau_{warming}$ mittels Streuung von mindestens einem weiteren Geräteparameter. Um diese Vermutung zu stützen, wurden weitere Simulationen durchgeführt, in denen bis auf einen Parameter alle fix waren. Der eine nicht-feste Parameter wurde dabei ähnlich wie in Abbildung 6.3 nacheinander gleichverteilt und normalverteil simuliert. Diese Prozedur wurde für alle Parameter durchgeführt, sodass für jeden Parameter untersucht werden konnte, wie er sich auf eine Population mit ansonsten festen Parameterwerten auswirkt. Es zeigt sich dort in jedem Fall die prognostizierte Einschwingphase, gefolgt von einem homogenen Verlauf. Eine zweite Testreihe untersuchte die Kombinationen, in denen zusätzlich immer die Starttemperatur gleichverteilt war, während ein weiterer Parameter gestreut wurde. Auch diese Simulationen bestätigten die Vermutung, indem sie alle einen homogenen Verlauf um den Mittelwert ohne Einschwingphase oder erkennbare Oszillation zeigten.
Die nächste zu klärende Frage ist jetzt, ob sich andere Resultate ergeben, wenn mehr als zwei Parameter gestreut werden. Da alle diese Parameter jedoch in diesem Kon-

text „nur“ die Zeiten $\tau_{cooling}$ und $\tau_{warming}$ beeinflussen, sollten sich keine abweichenden Erkenntnisse zeigen. Zur Klärung wurden wiederum Testreihen mit variierenden Kombinationen von Streuungen durchgeführt. Hier ergab sich, dass wie erwartet bei einer vorliegenden gleichverteilten Starttemperatur sofort ein homogener Kurvenverlauf gegeben war. War die Starttemperatur fix und wurden zwei andere Parameter gestreut, so ergab sich jeweils das bekannte Bild einer abklingenden Oszillation der Einschwingphase, wobei die Länge und Intensität dieser Oszillation je nach gestreutem Parameterpaar unterschiedlich war. Dies bestätigt erneut die Vermutung.

Unter diesen Voraussetzungen kann nun die eigentliche Fragestellung dieses Kapitels erörtert werden: wie robust sind die Desynchronisationsmechanismen bei variierenden Parametern? Legt man jedoch die Abhängigkeit zu Grunde, dass durch Parameterstreuungen lediglich $\tau_{cooling}$ und $\tau_{warming}$ für jedes individuelle Gerät verändert werden (ersichtlich aus Formel 3.2), so müssten die Dämpfungsstrategien aus Kapitel 5 weiterhin korrekt arbeiten, da sie sich unter anderem nach genau diesen Zeiten richten. Auch hier wurden wiederum Testreihen durchgeführt, welche diese Vermutung bestätigen. Abbildung 6.4 zeigt exemplarische Ergebnisse für Simulationen mit TLR-Steuerung und zufallsbasierter Dämpfung, mit jeweils gleichverteilter Starttemperatur. Die restlichen Parameter waren bis auf die thermische Masse m_c normalverteilt. Die drei enthaltenen Kurven beschreiben die Möglichkeiten für m_c als Festwert, gleichverteilt und normalverteilt unter diesen Gegebenheiten. Es zeigt sich, dass zwar das Steuerungs- bzw. Lastverschiebungspotenzial der Kühlschrankpopulation durch die Variation der Parameter beeinflusst wird, die nachfolgende Dämpfung jedoch uneingeschränkt arbeitet.
Abschließend stellt Abbildung 6.5 Ergebnisse für Simulationen aller in Kapitel 4 vorgestellten Steuerungssignale in Verbindung mit den in Kapitel 5 entwickelten Dämpfungsstrategien dar. Für diese Testreihe waren alle Geräteparameter gestreut: die Starttemperatur war gleichverteilt, und die übrigen Parameter alle normalverteilt. Auch hier zeigt die Grafik, dass für jede Kombination aus Steuerungssignal und Dämpfungsstrategie eine vollständige Desynchronisation der Geräte erreicht werden konnte. Die erarbeiteten Dämpfungsstrategien sind also stabil und robust gegenüber Parametervariationen und -streuungen.

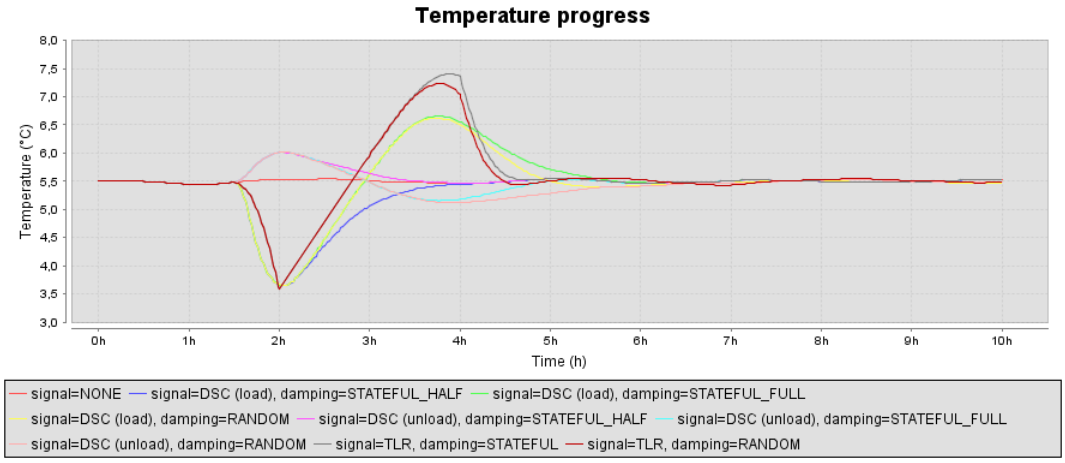


(a)

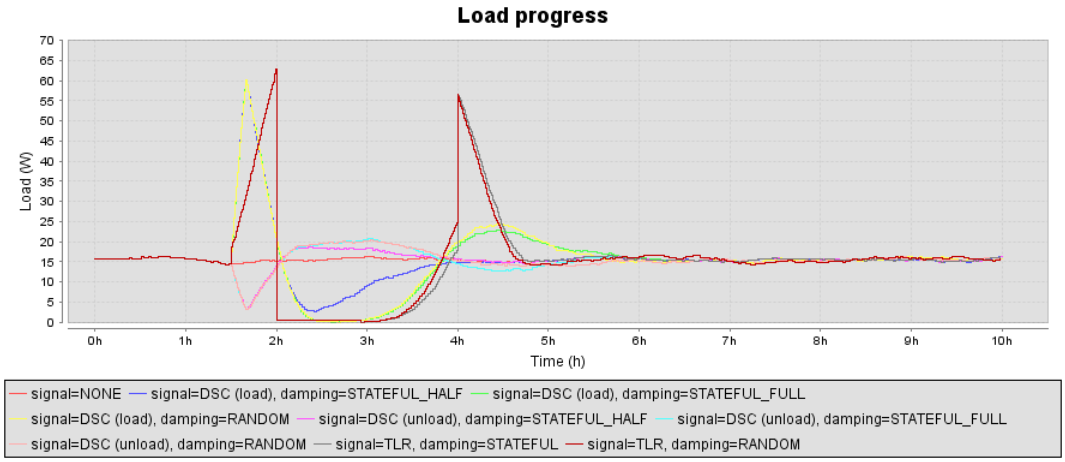


(b)

Abbildung 6.4: Auswirkungen von Parametern auf Signal TLR und zufallsbasierte Dämpfung.



(a)



(b)

Abbildung 6.5: Übersicht der Signale und Dämpfungen bei gestreuten Parametern.

6.3 Externe Einflüsse

Nachdem nun die globalen Simulationseinstellungen sowie diverse Variationen der Geräteparameter untersucht wurden bleibt noch die Frage zu klären, inwiefern externe Einflüsse Auswirkungen auf die Zustandsverteilung der Geräte haben können. Mit externen Einflüssen ist dabei die Bedienung durch den Anwender gemeint, denn jedes Mal, wenn der Anwender die Kühlschranktür öffnet, erhöht sich einerseits die Last, da das Kühlschrankinnere für den Öffnungszeitraum beleuchtet wird, während andererseits wahrscheinlich die thermische Masse verändert wird, indem etwas aus dem Gerät herausgenommen oder zusätzlich hineingestellt wird. Weiterhin wird die Innentemperatur durch die geöffnete Tür und dem damit einhergehenden Luftaustausch mit der Umgebung beeinflusst.
Diese Überlegungen können in dieser Arbeit nur am Rande betrachtet werden, daher wurde auf ein physikalisches Luftaustauschmodell verzichtet und der Fokus auf die ersten beiden Punkte gelegt. Im privaten Umfeld des Autors wurde dazu eine einwöchige Umfrage durchgeführt, in welcher eine grobe Auflistung der Öffnungszeiträume von Kühlschränken in Privathaushalten entstanden ist. Aufgrund des Umfanges musste hier leider die Messung der Veränderung der thermischen Masse vernachlässigt werden.
Im Endeffekt kann hier also leider nur die Auswirkung der erhöhten Last durch die Innenbeleuchtung bei einer geöffneten Tür erörtert werden. Die entstandene Stichprobe besteht aus sieben Haushalten mit durchschnittlich zwei Personen, und liefert die in Tabelle 6.6 dargestellten Mittelwerte. Die Werte in den Zellen bezeichnen dabei die Anzahl an Öffnungen innerhalb des angegebenen Zeitraumes und für die entsprechende Dauer. Die Werte wurden nun im Folgenden weiter vereinfacht, indem zunächst die drei Spalten zu einer einzelnen zusammengefasst wurden. Dazu wurde für jede Zeile der Mittelwert gebildet, wobei die erste Spalte einfach, die zweite Spalte zweifach und die dritte Spalte vierfach gewertet wurde, sodass der gebildete Wert in etwa die Anzahl der Öffnungen mit einer Dauer von 5 Sekunden widerspiegelt. Anschließend wurden diese 21 Werte durch erneute Bildung eines Mittelwertes zusammengefasst, wodurch sich $\bar{x} = 11.07$ ergab. Ganz grob gesehen lässt sich also hier sagen, dass ausgehend von der vorliegenden Stichprobe im Mittel 11.07 Öffnungen pro Tag mit einer Dauer von durchschnittlich 5 Sekunden erfolgen.
Um nun diese Erkenntnis in die Simulation einfließen zu lassen wurde in der Software eine neue Klasse Lamp angelegt, die ähnlich wie der Controller auf ein AbstractFridge Objekt aufsetzt und es modifiziert. Die Modifikation beschränkt sich

Dauer der geöffneten Tür:		**<5sek**	**5-15sek**	**>15sek**
Montag	Vor 10:00	1.6	2	0
	10:00-16:00	3.4	2.75	2
	Nach 16:00	5.83	2.4	1.5
Dienstag	Vor 10:00	2.67	1	1
	10:00-16:00	3,6	1	1
	Nach 16:00	4.5	2.5	1
Mittwoch	Vor 10:00	2.2	1.75	1
	10:00-16:00	2.2	1.33	1
	Nach 16:00	4.14	1.6	1.8
Donnerstag	Vor 10:00	2.2	1.5	1.5
	10:00-16:00	2	2.67	0
	Nach 16:00	4.8	2.83	2
Freitag	Vor 10:00	3.4	1.5	1
	10:00-16:00	4.5	3	2
	Nach 16:00	5.14	2.5	1.75
Samstag	Vor 10:00	2.25	1	0
	10:00-16:00	4	2.67	0
	Nach 16:00	4.4	3.4	3.5
Sonntag	Vor 10:00	3.33	1.5	1
	10:00-16:00	3.86	2.17	1.5
	Nach 16:00	7	3.8	1.5

Tabelle 6.6: Mittelwerte der empirischen Kühlschrank-Öffnungszeiträume.

hier allerdings darauf, die Last des Kühlgerätes um einen gleichverteilten Wert im Intervall $[5\,W, 25\,W]$ zu bestimmten Zeitpunkten zu erhöhen (Tür geöffnet, Lampe an) bzw. zu erniedrigen (Tür wieder geschlossen, Lampe aus). Die Zeitpunkte richten sich dabei nach dem oben errechneten Mittelwert: legt man eine mögliche Zeitspanne von 18 Stunden zu Grunde (06 Uhr - 24 Uhr), so erfolgt im Mittel pro Gerät alle $\frac{18.0 \cdot 60.0}{\bar{x}}$ Minuten eine Öffnung. Um diese Zeitpunkte zu streuen wurde zusätzlich die Standardabweichung der obigen 21 Werte ermittelt, sie beträgt $\sigma = 4.81$. Eine simulierte Lampe erhöht nun alle x Minuten für 5 Sekunden die Last des Kühlgerätes, wobei der Wert x eine gleichverteilte Zufallszahl aus dem Intervall $[\bar{x} - \sigma, \bar{x} + \sigma]$ ist. Eine Simulation ergibt die in Abbildung 6.6 dargestellten Kurven. Die grüne Kurve

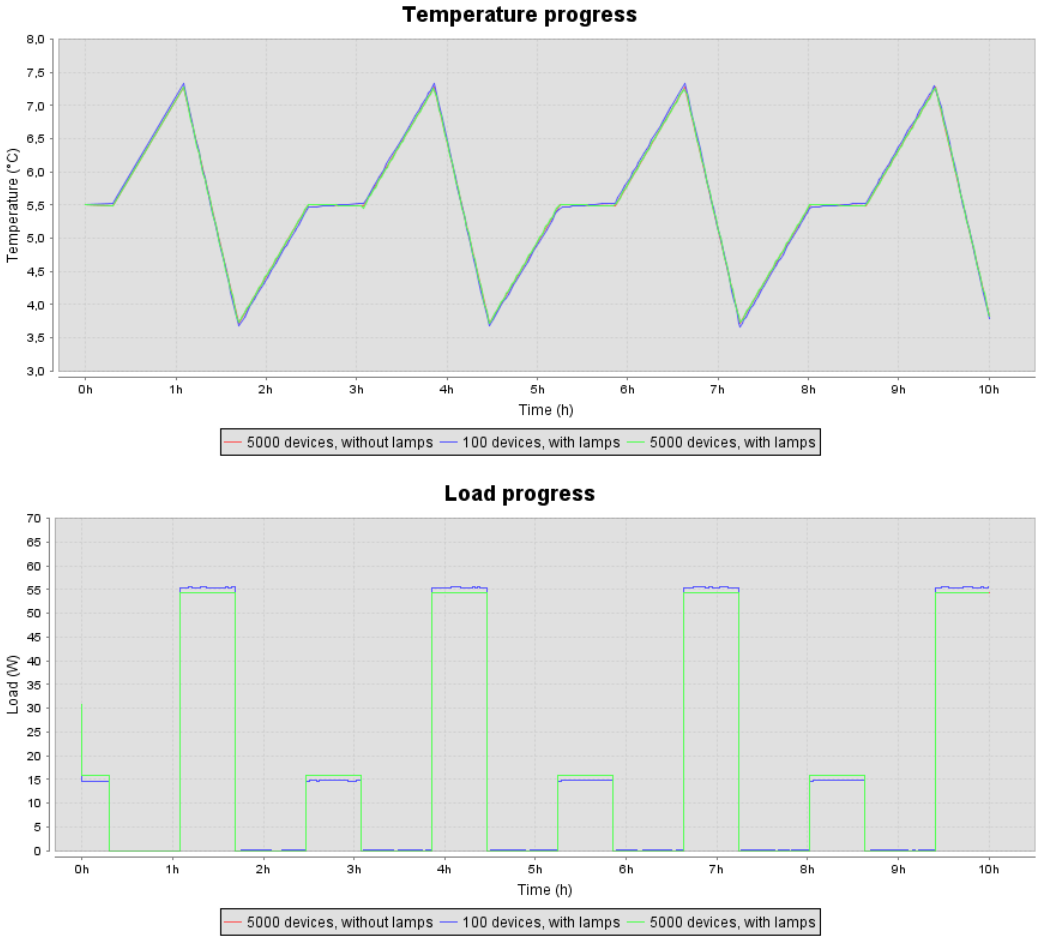


Abbildung 6.6: Auswirkungen von simulierten Innenbeleuchtungen.

bezeichnet eine Simulation mit den in diesem Kapitel ermittelten Parametern und aktivierter Simulation der Lampen. Die rote Kurve zeigt die gleiche Simulation ohne Lampen. Die Unterschiede zwischen diesen beiden Simulationsläufen sind so gering, dass die grüne Kurve die rote vollständig überdeckt. Um Fehler auszuschließen wurde eine dritte Kurve eingeblendet, welche einen Simulationslauf mit nur 100 Geräten und aktivierten Lampen darstellt. Hier sind schon deutlichere Unterschiede erkennbar, was an der geringeren Streuung durch die kleinere Populationszahl liegt.
Es lässt sich somit abschließend festhalten, dass der Einfluss von Innenbeleuchtungen keine Rolle für Simulationen in den hier verwendeten Größenordnungen spielt und vernachlässigt werden kann. Daher wird auch keine genauere Aufschlüsselung nach Tageszeiten oder Wochentagen der erhobenen Stichprobe durchgeführt. Inwieweit die weiteren eingangs erwähnten Einflussfaktoren der veränderten thermischen Masse bzw. Wärmeaustausch durch Luft zu beachten sind kann an dieser Stelle leider nicht beurteilt werden und müsste bei Bedarf genauer untersucht werden.

Kapitel 7

Controller Design

Im vorigen Kapitel wurde gezeigt, dass die entwickelten Desynchronisationsstrategien robust gegenüber Parametervariationen sind. Um die gewonnenen Erkenntnisse leicht weiterverarbeiten und in einer realen Hardwareumgebung testen zu können, müssen die vorgestellten Controllervarianten jedoch noch genauer spezifiziert werden. Die Simulationsmodelle wurden nach dem Paradigma des *Event Graph Modeling* erstellt. In Abschnitt 3.1 ist bereits ein einfacher Vergleich zwischen konventionellen Zustandsdiagrammen und den Ereignisgraphen vorgenommen worden. Im hardwarenahen Controllerentwurf würde ein Entwickler nun wahrscheinlich ein Synthesewerkzeug zur Transformation der Modelle in konkrete Automaten bzw. Schaltungen zur Realisierung in programmierbaren Hardwarebausteinen verwenden. Für die vorliegende Arbeit wird es jedoch ausreichen, die Modelle vom ereignisorientierten Paradigma vorbereitend in Modelle von Automaten nach dem zustandsbasierten Paradigma zu übersetzen. Diese noch recht abstrakten Automaten können dann später bei Bedarf einfach in konkrete Schaltungen umgesetzt werden. Um eine möglichst kompatible Darstellung zu gewährleisten, werden die Modelle in der *Statechart* Notation (siehe [17]) angefertigt, welche ebenfalls in den bekannten Synthesewerkzeugen zum Einsatz kommt. Im Besonderen wird hier die auf UML basierende Darstellung aus [12][1] verwendet. Eine allgemeingültige Vorschrift zur Transformation der Graphen konnte leider nicht ausfindig gemacht werden, daher muss von Modell zu Modell einzeln entschieden werden, wie die Umsetzung vorgenommen wird. Ein prinzipieller

[1] aus http://www.embedded.com/1999/9901/9901feat1.htm, zuletzt besucht: 24. Oktober 2008

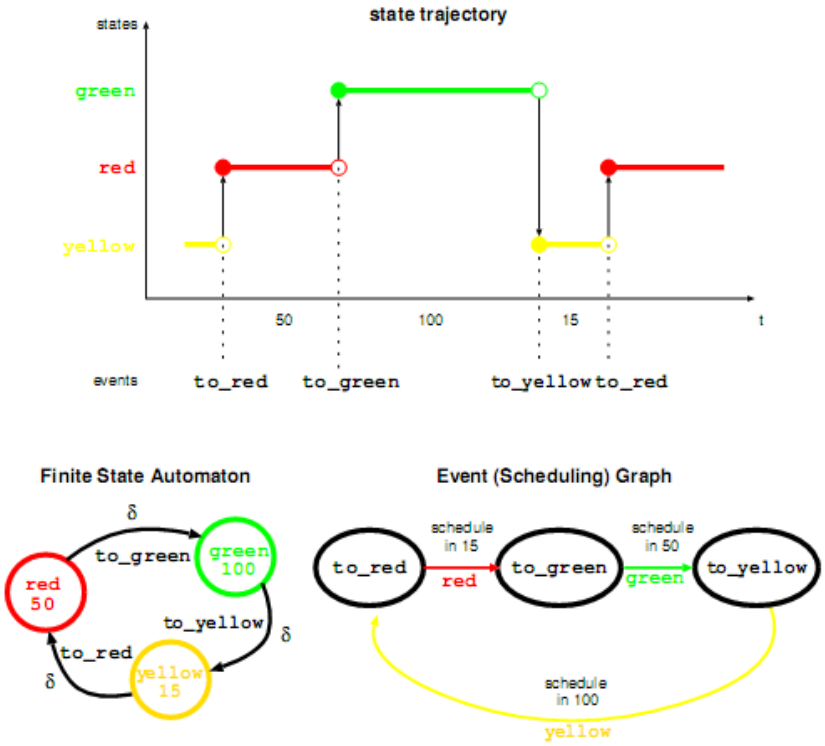


Abbildung 7.1: Zusammenhang zwischen Zustandsautomaten und Ereignisgraphen, entnommen aus [36].

Anhaltspunkt ist in [36, Abs. 2][2] gegeben, dort wird anhand der in Abbildung 7.1 reproduzierten Grafik der Zusammenhang zwischen den beiden Paradigmen erläutert. Im unteren Teil der Abbildung ist das Modell einer Ampelschaltung in beiden Paradigmen visualisiert. Im oberen Teil befindet sich eine Darstellung des zeitlichen Verlaufs dieser Schaltung im Betrieb. Offenbar zeigen die Modelle eine duale Sichtweise auf den zeitlichen Verlauf der Schaltung. Während in der Darstellung als Zustandsautomat die Zustände des Systems im Vordergrund stehen, und Änderungen des Systems als Transitionen in andere Zustände modelliert werden, sind in dem Ereignisgraphen primär die Ereignisse dargestellt, die zu solchen Änderungen führen. Eine Abbildung der Paradigmen aufeinander könnte dadurch identifiziert

[2] aus http://www.cs.mcgill.ca/~hv/classes/MS/discreteEvent.pdf, zuletzt besucht: 24. Oktober 2008

werden, dass die Pfeile und die Kreise ihre Semantik tauschen: in Richtung Ereignisgraph → Zustandsautomat werden Ereignisse zu Transitionen zwischen Zuständen, und die Wartezeiten zwischen den Ereignissen können als Zustände dargestellt werden. Solch eine Transformation berücksichtigt zwar noch nicht die logische Abhängigkeit zwischen einzelnen Zustandsübergängen bzw. Ereigniseinplanungen, kann aber als Basis für eine Übersetzung der in dieser Arbeit entwickelten Ereignisgraphen dienen.

7.1 Basiseinheit

Als erstes wird die grundlegende Funktionalität des Kühlgerätes betrachtet. Diese wurde im Verlauf dieser Arbeit durch insgesamt drei Modellvarianten beschrieben: iterativ (Abschnitt 3.3.1), linear (Abschnitt 3.4.1) und abschließend noch einmal linear in einer kompakteren Darstellung (Abschnitt 3.4.2). Die linearen Varianten wurden aus Gründen der einfacheren Berechenbarkeit erstellt. Um die Komplexität eines realen Controllers gering zu halten bietet es sich an, eine dieser linearen Varianten als Basis zur Transformation zu verwenden. Abbildung 7.2 zeigt den resultierenden zustandsbasierten Automaten in der *Statechart* Notation. Die Funktionsweise ist fast identisch zum nicht-kompakten Ereignisgraphen des linearen Modells, die Abweichungen werden im Folgenden erläutert: bei Eintritt in einen der Zustände cooling oder warming wird zunächst das Kühlaggregat aktiviert / deaktiviert. Anschließend wird die aktuelle Temperatur ermittelt, in T_{from} gespeichert und die benötigte Zeitspanne τ_{switch} zum Erreichen der Zieltemperatur T_{dest} berechnet. Die Berechnung erfolgt mittels Formel 3.6 und stützt sich auf den zuletzt bekannten Wert für $\tau_{cooling}$ bzw. $\tau_{warming}$. Da im realen Betrieb der Kühlschrankinhalt (also der Wert m_c) im Laufe der Zeit unvorhersehbar variiert, sind die Zeitspannen $\tau_{cooling}$ und $\tau_{warming}$ nicht mehr fest definiert. Daher wurde ein Zähler counter eingeführt, der die Verweildauer in den Zuständen misst und bei Austritt aus dem Zustand, basierend auf dem Intervall $[T_{from}, T_{current}]$, den Wert $\tau_{cooling}$ bzw. $\tau_{warming}$ aktualisiert. Damit diese Zeitmessung funktioniert, ist in den Zuständen jeweils ein Unterzustand enthalten, der in periodischen Abständen (hier: eine Sekunde) erneut betreten wird. Dabei wird jedes Mal der Zeitzähler inkrementiert. Sobald die Verweildauer des aktuellen Zustands die errechnete Zeitspanne τ_{switch} erreicht, wird in den komplementären Zustand mit der passenden Zieltemperatur gewechselt. Dies geschieht auch dann, wenn vorher die Grenztemperatur T_{min} bzw. T_{max} unter-/überschritten wird. Diese Abfrage dient der Sicherheit, falls die lineare Berechnung nicht exakt genug ist, und

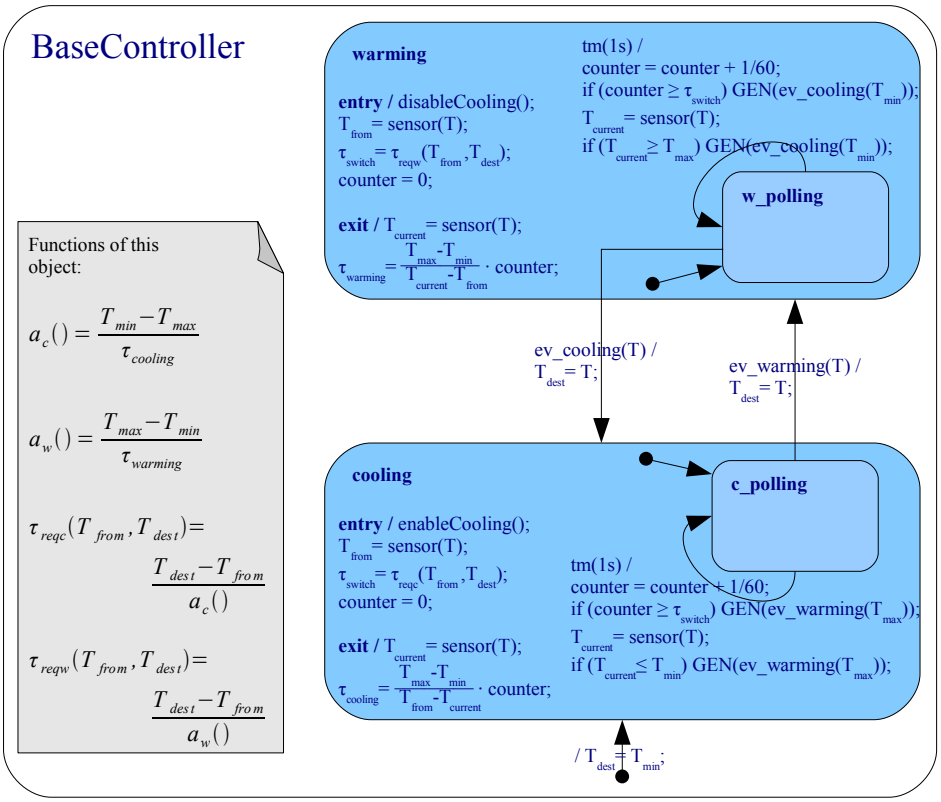


Abbildung 7.2: Zustandsautomat des Kühlschrankmodells.

garantiert die Einhaltung der geforderten Temperaturgrenzen.

Dieser Automat dient als Basis für die weiteren Zustandsgraphen. Die folgenden Automaten sind so entworfen, dass sie als unabhängige Erweiterung zu sehen sind, das heißt sie laufen parallel zu obiger Schaltung, und greifen bei Bedarf durch die Aufrufe BaseController.GEN(ev_cooling(T)) und BaseController.GEN(ev_warming(T)) in den normalen Arbeitsablauf des Basiscontrollers ein und sind damit in der Lage, zu beliebigen Zeitpunkten einen Phasenwechsel hervorzurufen. Die Parallelität der Schaltungen bedeutet zwar an dieser Stelle einen etwas aufwändigeren Schaltungsaufbau, da mehrere Module gleichzeitig aktiv sein können, bietet jedoch wegen der daraus resultierenden Unabhängigkeit auch eine enorme Flexibilität in Bezug auf Entwicklung und Wartung. So können auf diese Weise beispielsweise sehr leicht weitere Module entwickelt und hinzugefügt werden, da die hierarchisch unter ihnen

liegenden Module dafür nicht modifiziert werden müssen. Wie in den folgenden Abschnitten deutlich wird, ist es somit auch kein Problem, die separaten Erweiterungen für die DSC- und für die TLR-Steuerung gleichzeitig zu betreiben.

7.2 Controller-Erweiterung: DSC

Der folgende Automat stellt die Erweiterung für die DSC-Steuerung dar. Abbildung 7.3 zeigt den Zustandsgraphen des Automaten. Wie bereits beschrieben ar-

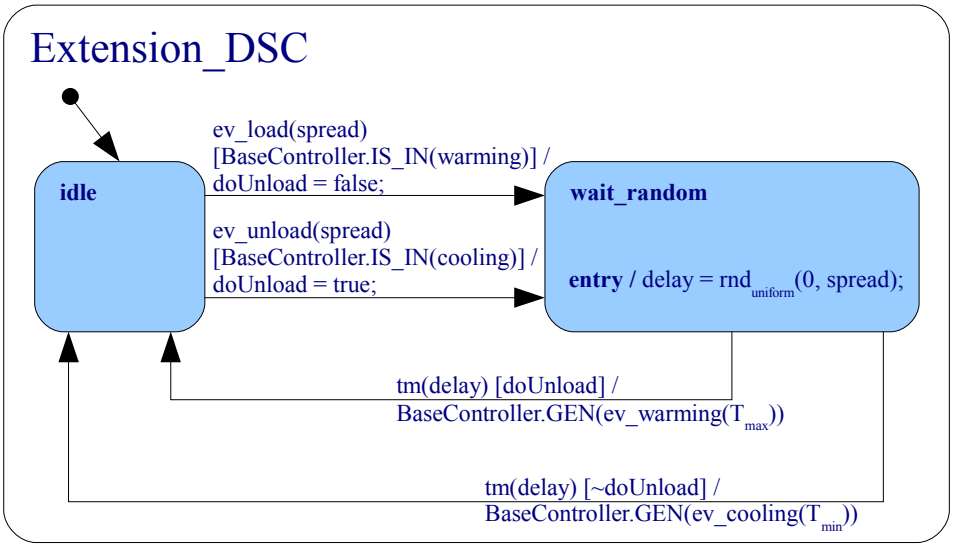


Abbildung 7.3: Zustandsautomat der DSC-Erweiterung.

beitet dieser Automat parallel zur Basiseinheit. In der Ausgangsstellung befindet er sich im Zustand idle und wartet auf einkommende DSC-Signale. Tritt solch ein Signal ev_load(spread) oder ev_unload(spread) auf, so wird zunächst überprüft, ob sich das Kühlgerät bereits in der angeforderten Phase befindet. Falls nicht, wird die geforderte Phase durch Setzen der bool'schen Variable doUnload gespeichert und in den Zustand wait_random gewechselt. Dort wird eine gleichverteilte Zufallszahl aus dem Intervall $[0, spread]$ ermittelt und die resultierende Anzahl an Zeiteinheiten gewartet. Nach Ablauf der Verzögerung wird abhängig vom Wert in doUnload eines der Signale ev_warming(T_{max}) oder ev_cooling(T_{min}) an die Basiseinheit gesendet, welche dann entsprechend die Phase wechselt und die übergebene Zieltemperatur ansteuert. Das Modul hat derweil wieder in den idle Zustand gewechselt und wartet

auf das nächste Signal.
Da dieses Modul so kompakt ist, wurde auf die Modellierung der Desynchronisationsstrategien hier als eigenständige Module verzichtet, sondern stattdessen in beiden Varianten in dieses Modul integriert. Abbildung 7.4 zeigt das Modul mit integrierter zustandsbehafteter Dämpfung. Im Vergleich zu obigem Automaten ist lediglich

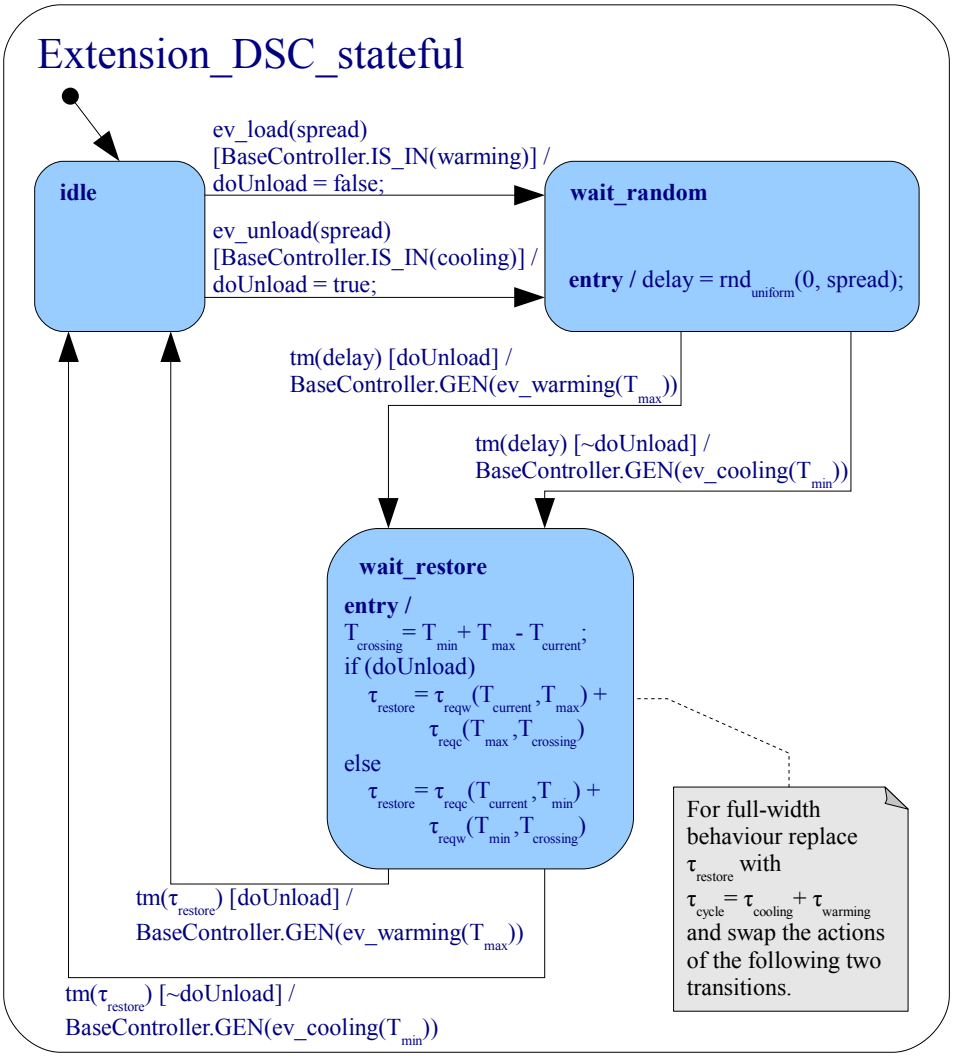


Abbildung 7.4: Zustandsautomat der DSC-Erweiterung mit zustandsbehafteter Dämpfung.

der Zustand wait_restore hinzugekommen, der nach dem Senden des Signales an den Basiscontroller den Zeitpunkt des nächsten Einschreitens zur Zustandswiederherstel-

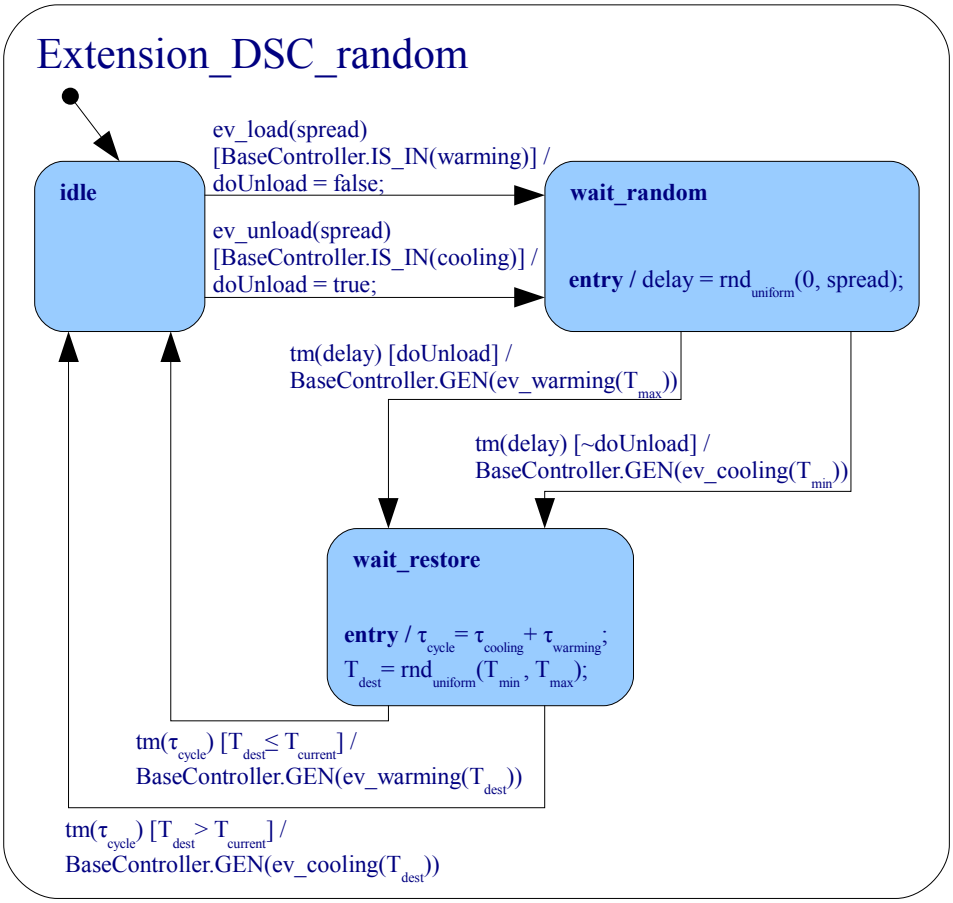


Abbildung 7.5: Zustandsautomat der DSC-Erweiterung mit zufallsbasierter Dämpfung.

lung errechnet und dieses Intervall abwartet, bevor das entsprechende Signal wiederum an die Basiseinheit gesendet wird. Die Arbeitsweise ist identisch zu dem Ereignisgraphen aus Abschnitt 5.1. Die abgebildete Version errechnet den Wert $\tau_{restore}$ entsprechend der ersten Version der DSC-Zustandswiederherstellung (vgl. Listing 5.2 und Abbildung 5.3). Um das Verhalten der zweiten Version (siehe Abbildung 5.4) zu erreichen, muss der Wert $\tau_{restore}$ einfach durch $\tau_{cycle} = \tau_{cooling} + \tau_{warming}$ ersetzt werden.

Analog dazu ist in Abbildung 7.5 das Modul mit integrierter zufallsbasierter Dämpfung dargestellt. Auch hier ist nur ein weiterer Zustand hinzugekommen, der sich ebenfalls zeitlich direkt nach der Modifikation des Basiscontrollers befindet. Die

Funktionsweise ist bekannt: nach τ_{cycle} Zeiteinheiten wird eine gleichverteilt ermittelte Zufallstemperatur aus dem Intervall $[T_{min}, T_{max}]$ als Zieltemperatur an die Basiseinheit gesendet, welche dann entsprechend ihre Phase anpasst. Anschließend verharrt das Modul wieder im Zustand idle, bis das nächste Steuersignal eintrifft.

7.3 Controller-Erweiterung: TLR

Die Erweiterung für die TLR-Steuerung ist ebenso wie das Modul der DSC-Steuerung als parallele Komponente entworfen. Es wurde die in Abschnitt 4.2.1 verwendete Strategie des *optimalen Kühlens* realisiert. Da diese Strategie jedoch relativ viele Fallunterscheidungen und Berechnungen bedingt (vgl. Abbildung 4.8), wird der Zustandsautomat hier vereinfacht dargestellt. Abbildung 7.6 zeigt die resultierende Grafik. Die vollständige Version mit detaillierten Zustandsbeschreibungen und allen Transitionen ist in Anhang A zu finden, dort sind außerdem die hier verwendeten

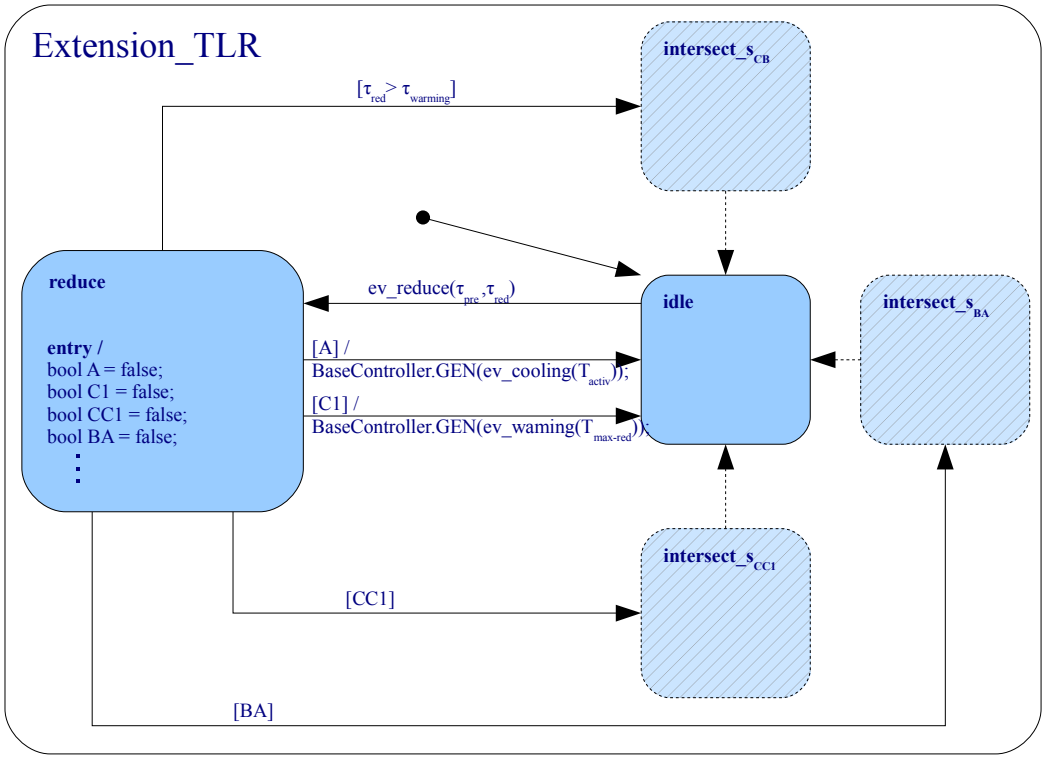


Abbildung 7.6: Zustandsautomat der TLR-Erweiterung (vereinfacht).

Funktionen calculate_$\tau_{\gamma_{s_{xx}}}$ beschrieben. Wie auch bei den anderen Erweiterungsmodulen startet dieser Automat im Zustand idle. Trifft nun das Signal ev_reduce() ein, so wechselt die Schaltung in den Zustand reduce und ermittelt dort anhand der Signalparameter $\tau_{preload}$ und τ_{reduce} sowie der aktuellen Geräteparameter $\tau_{cooling}$ und $\tau_{warming}$, in welche Klasse der Bereichseinteilung aus Abschnitt 4.2 die aktuelle Situation eingeordnet wird, und wechselt in den passenden der vier möglichen Folgezuständen. In den drei hier schraffiert dargestellten Zuständen wird durch die bereits bekannte Schnittpunktberechnung mit den Trenngeraden des klassifizierten Bereiches die konkrete Modifikation des Kühlvorganges geplant und nach einer entsprechenden Wartezeit durchgeführt. Die einzelnen Möglichkeiten dieser Planung und Modifikation wurden in Bezug auf Abbildung 4.8 bereits erläutert und sind wiederholend noch einmal in der detaillierten Abbildung dieses Automaten im Anhang der Arbeit einzusehen. Nach der erfolgten Modifikation wechselt der Automat wieder in den Zustand idle und ist bereit für ein weiteres TLR-Steuerungssignal. Allerdings können zwei Sonderfälle auftreten. Der erste betrifft den Fall, dass das Gerät in die Klasse A eingeteilt wird. Unter diesen Umständen kann das Gerät die erforderliche Grenztemperatur $T_{max-act}$ nicht mehr erreichen und muss ab sofort bis zum Beginn des Reduktionsintervalles kühlen, um dieses möglichst lang zu überstehen. Dieser Fall wird durch die Transition beschrieben, die direkt aus dem Zustand reduce zurück nach idle führt und das Signal ev_cooling(T_{activ}) sendet. Der zweite Sonderfall tritt dann ein, wenn das Gerät in die Klasse C1 eingeteilt wird. Hier ist die Temperatur bereits so gering, dass überhaupt nicht mehr gekühlt werden muss, um das Reduktionsintervall zu überstehen, es muss sofort in die Aufwärmphase gewechselt werden. Als Ziel wird die Temperatur $T_{max-red}$ angegeben, es handelt sich um die Temperatur, die das Gerät unmittelbar nach dem Reduktionsintervall erreicht haben wird. Nach der hier verfolgten Strategie des *optimalen Kühlens* soll das Gerät direkt nach der Lastreduktion den Kühlvorgang beginnen, dies wird in diesem Sonderfall durch ebendiese Zieltemperatur erreicht. Ausgeführt wird dieser Fall durch die zweite Transition vom Zustand reduce in Richtung idle.

Im Gegensatz zum vorherigen Abschnitt müssen die Erweiterungen für die Desynchronisationsstrategien nicht direkt in dieses Modul integriert werden. Da dieses Modul sich allein nach den mit dem Signal übergebenen Parametern sowie den allgemein einsehbaren Geräteparametern richtet, können die Algorithmen zur Dämpfung als separate Module entwickelt werden, welche dann ebenfalls parallel zu den bisherigen Modulen arbeiten. In Abbildung 7.7 ist die zustandsbehaftete Dämpfung

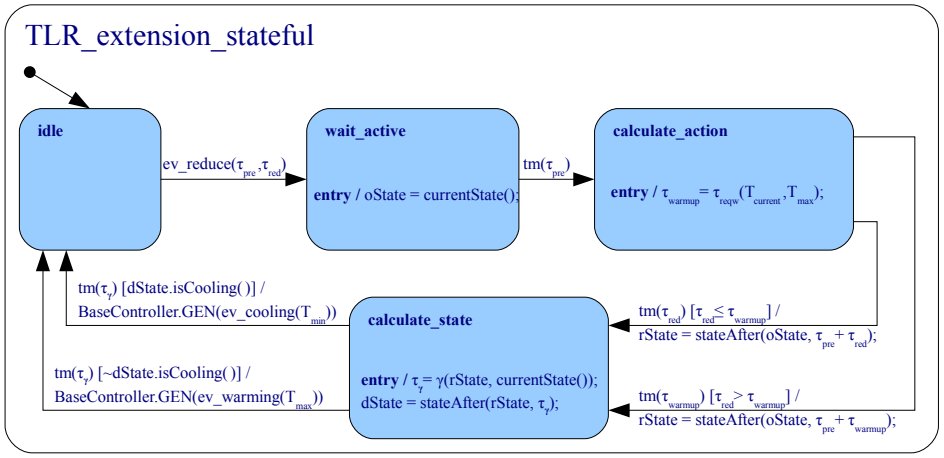


Abbildung 7.7: Zustandsautomat der zustandsbehafteten Dämpfung für die TLR-Steuerung.

als separates Modul dargestellt. Wie gehabt befindet sich die Schaltung solange im Zustand idle, bis das erwartete Signal ev_reduce() eintrifft. Analog zu dem in Abbildung 5.5 vorgestellten Mechanismus wechselt der Automat nun in den Zustand wait_active, speichert dort den aktuellen Gerätezustand in oState und bleibt dort durch die Anweisung tm(τ_{pre}) bis zum Beginn des Reduktionsintervalles im Zeitpunkt t_{activ} stehen. Zu diesem Zeitpunkt wechselt die Schaltung dann in den Folgezustand calculate_action und ermittelt dort den Gerätezustand, den das Kühlgerät ohne eine Modifikation durch das TLR-Signal jetzt hätte und speichert ihn in rState. Nun wartet der Automat, bis das Reduktionsintervall vorüber ist oder bis das Gerät seine Maximaltemperatur erreicht hat (je nachdem was von beidem eher eintrifft), und wechselt dann in den Zustand calculate_state. Hier wird nun der nächstgelegene Schnittpunkt γ des ausgehend vom gespeicherten Gerätezustand rState anzunehmenden regulären und des aktuellen Temperaturverlaufes errechnet, und der Gerätezustand dState ermittelt, den das unmodifizierte Gerät in diesem Zeitpunkt hätte (vgl. Abbildung 5.1). Nun wird wiederum bis zu dem Zeitpunkt dieses Schnittpunktes abgewartet und dann ein entsprechendes Signal an den Basiscontroller gesendet, damit dieser in den korrekten Zustand wechselt und ab nun wieder dem errechneten Temperaturverlauf aus dState folgt. Damit ist die Zustandswiederherstellung abgeschlossen und das Modul befindet sich erneut im Zustant idle.
Die hier verwendeten Funktionen currentState(), stateAfter() und $\gamma()$ enthalten zum Teil recht umfangreiche Berechnungen und wurden daher nicht direkt im Zustands-

diagramm erläutert. Sie sind stellvertretend als Code-Fragmente in Anhang B enthalten.
Auch die zufallsbasierte Dämpfung kann als separates Modul erstellt werden, Abbildung 7.8 zeigt den resultierenden Automaten. Ähnlich zu den vorangegangenen

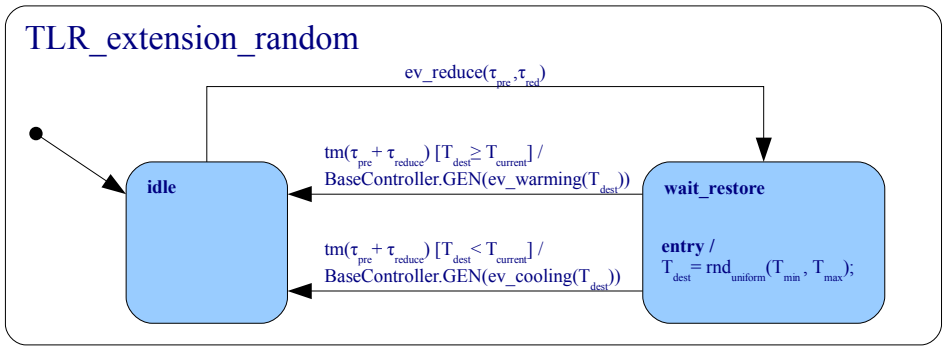


Abbildung 7.8: Zustandsautomat der zufallsbasierten Dämpfung für die TLR-Steuerung.

Modulen startet auch diese Schaltung im Zustand idle und wartet auf das TLR-Steuerungssignal. Trifft es ein, wird in den Zustand wait_restore gewechselt, wo zunächst eine gleichverteilt berechnete Zufallstemperatur aus dem Intervall $[T_{min}, T_{max}]$ ermittelt wird. Nach einer Verzögerung von $\tau_{pre} + \tau_{reduce}$ (entspricht dem Ende des Reduktionsintervalles) wird diese dann mit dem passenden Signal als Zieltemperatur an den Basiscontroller gesendet, welcher dann entsprechend seinen Zustand anpasst. Damit wurde das betreffende Gerät in einen zufällig gewählten Temperaturverlauf geschoben und das Modul befindet sich wieder im Startzustand idle.

7.4 Validierung der Modelle

Die Überführung der Ereignisgraphen der in diesem Kapitel entwickelten Automaten wurde ohne eindeutige Transformationsvorschrift durchgeführt, daher kann nicht ohne Weiteres auf die Fehlerfreiheit der Automatenmodelle geschlossen werden. Zur Validierung wurde deshalb erneut eine Software zur Simulation erstellt. In diesem Fall wurde auf keine existierende Bilbiothek zurückgegriffen, sondern ein eigenes kleines Java-Framework zur zeitdiskreten Simulation von Zustandsautomaten entwickelt. Dieses Framework besteht aus ca. 300 Zeilen Code in zehn Klassen und ist in Abbildung 7.9 als Klassendiagramm dargestellt. Unwichtige Funktionen wie *get*- und

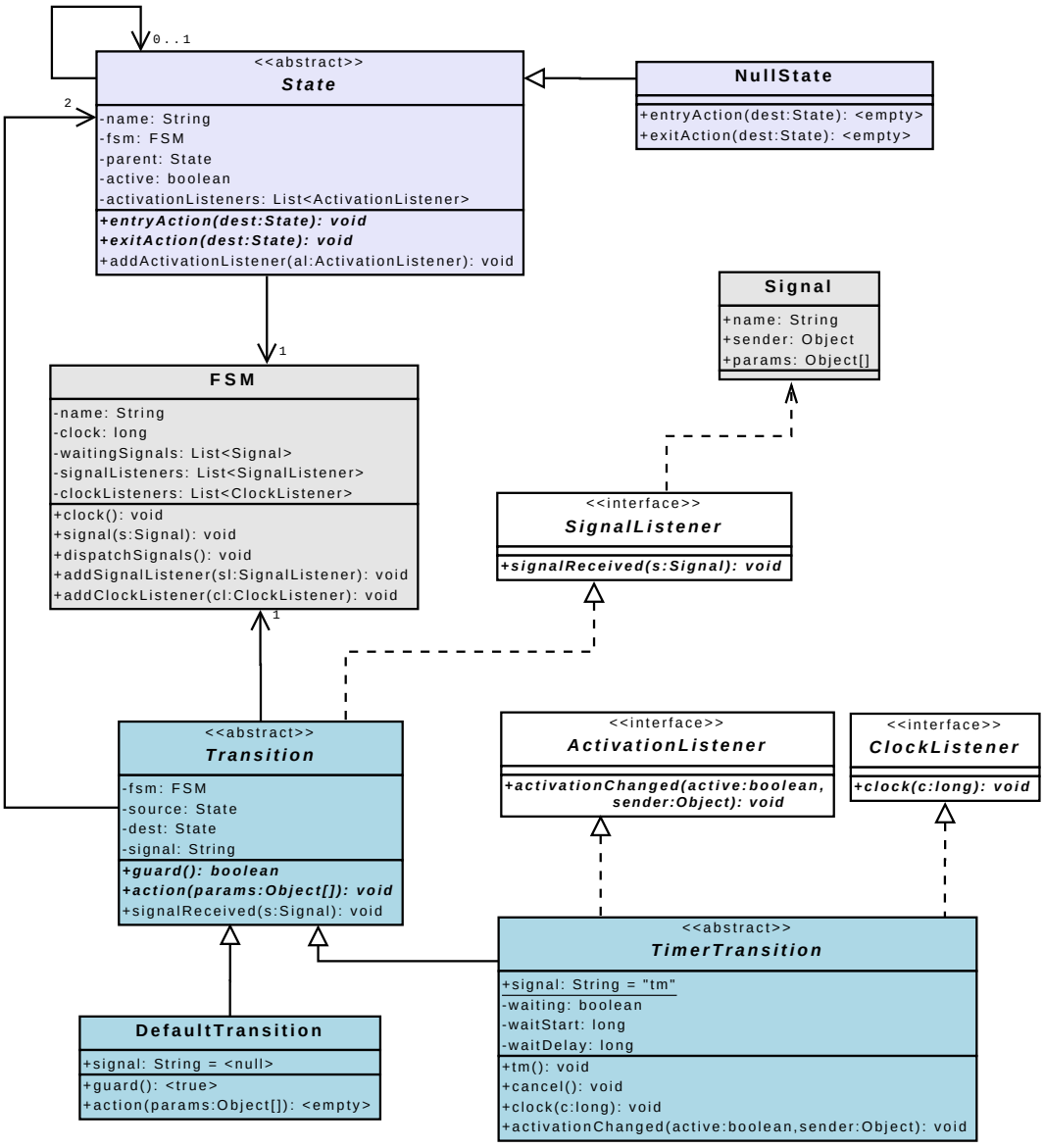


Abbildung 7.9: Klassendiagramm des FSM-Frameworks.

set-Methoden sind nicht eingezeichnet. Zentrale Klassen in diesem Framework sind FSM, State und Transition. Die Klasse FSM repräsentiert den gesamten endlichen Automaten und enthält die folgenden grundlegenden Methoden:

public void clock():
Stellt den Takt dar und wird von einer externen Referenzuhr regelmäßig aufgerufen. Bei jedem Aufruf wird zunächst das Feld clock inkrementiert, danach werden die registrierten ClockListener-Objekte benachrichtigt.

public void signal(Signal s):
Fügt das übergebene Signal in die Liste der wartenden Signale ein. Mittels dieser Methode kann dem Zustandsautomaten sowohl intern (etwa in dem aktuellen Zustand) als auch extern (z. B. durch eine Kommunikationsschnittstelle) ein Signal gesendet werden.

public void dispatchSignals():
Diese Methode wird in jedem Takt von dem gerade aktiven Zustand aufgerufen und bewirkt, dass die in der Warteliste enthaltenen Signale nacheinander an alle registrierten SignalListener-Objekte gesendet werden.

public void addSignalListener(SignalListener sl):
Registriert das übergebene SignalListener-Objekt.

public void addSignalListener(ClockListener cl):
Registriert das übergebene ClockListener-Objekt.

Diese Klasse ist also als organisatorisches Zentrum anzusehen, da hier der Takt des Automaten ausgelöst sowie Signale empfangen werden. Beides wird dann an die angeknüpften Objekte weitergeleitet.

Die Klasse State repräsentiert einen Zustand und unterliegt einer hierarchischen Struktur, kann also gemäß [17] mehrere Unter- sowie einen Überzustand haben. Zustände werden durch ihnen zugeordnete Transitionen aktiviert oder deaktiviert und führen dabei die entsprechende Methode entryAction(State dest) bzw. exitAction(State dest) aus. Bei jeder (De-)Aktivierung werden die registrierten ActivationListener-Objekte benachrichtigt. Wie in der Beschreibung der Klasse FSM bereits erwähnt, sorgt der gerade aktive Zustand dafür, dass pro Takt jeweils die wartenden Signale des zugeordneten FSM-Objektes abgearbeitet werden. Die abgeleitete Klasse NullState repräsentiert einen initial aktiven, aber leeren Zustand ohne Aktionen und

existiert aus programmiertechnischen Gründen.

Die Klasse Transition stellt abschließend die Transitionen des Zustandsgraphen dar. Sie ist die komplexeste Klasse dieses Frameworks, da sie die aktivste Komponente des Automaten beschreibt. Transitionen implementieren alle das Interface SignalListener, da ein Zustandübergang optional durch ein Signal ausgelöst werden kann. Weiterhin kann eine Transition einen Wächter besitzen, der die Ausführung des Überganges unter eine Bedingung stellt. Dieser Wächter wurde durch die abstrakte Methode guard() realisiert, welche einen bool'schen Wert zurückliefert. Eine weitere abstrakte Methode ist action(Object[] params), sie enthält in der konkreten Implementierung optional die auszuführenden Aktionen der Transition. Der Parameter params beschreibt ein Feld von Werten, die mit einem eventuellen auslösenden Signal übergeben wurden. Transitionen werden auf zwei Arten aktiv. Entweder lauschen sie auf ein bestimmtes Signal und erhalten dieses irgendwann über den Listener-Mechanismus mittels der Methode signalReceived(Signal s), oder sie stellen sogenannte *null transitions* dar. Diese warten auf kein spezielles Signal und werden nur beim Betreten des Ursprungszustandes abgearbeitet. Enthält ein Zustand eine ausgehende Transition ohne Signal, so wird diese Transition sofort bei Betreten des Zustandes weiterverfolgt, sofern der assoziierte Wächter dies erlaubt. Eine spezielle Form dieser *null transition* ist die Klasse DefaultTransition. Diese repräsentiert den Standard-Übergang, der beispielsweise beim Starten des Automaten verfolgt wird (in den Zustandsdiagrammen durch einen ausgefüllten Kreis am Startpunkt der Transition gekennzeichnet). Eine weitere spezielle Form des Zustandsüberganges ist die Transition mit Verzögerung. Diese wird durch die Klasse TimerTransition realisiert und implementiert die Interfaces ActivationListener und ClockListener. Eine solche Transition wird benachrichtigt, sobald ihr Ursprungszustand aktiviert wird. Daraufhin wird die Methode tm() aufgerufen, welche den internen Zeitzähler aktiviert. Über das ClockListener Interface wird die Transition bei jedem Takt erneut benachrichtigt. Sind nach einer Weile gemäß des internen waitDelay-Wertes genügend Takte vergangen und der Ursprungszustand noch immer aktiv, so wird die Transition ebenfalls aktiv und der Zustandsübergang wird durchgeführt. Über die Methode cancel() kann der Zeitzähler einer solchen Transition auch vor Ablauf wieder deaktiviert werden, diese Methode existiert ebenfalls aus programmiertechnischen Gründen.

Mittels dieses kleinen Frameworks können nun die in diesem Kapitel vorgestellten Zustandsdiagramme simuliert und auf ihre korrekte Funktionsweise hin über-

prüft werden. Dazu müssen die Diagramme lediglich durch Implementierung der Zustände und Transitionen in den jeweiligen Klassen in das Framework überführt werden. Da es sich bei den Automaten allerdings nur um die Controller handelt, die ein Kühlgerät steuern sollen, muss zusätzlich eine Klasse entwickelt werden, welche das Kühlgerät an sich repräsentiert. An dieser Stelle könnte natürlich auch ein *hardware-in-the-loop* Ansatz verfolgt und statt einer virtuellen Software-Klasse ein reales steuerbares Kühlgerät mit der Simulation gekoppelt werden. Für die schnelle Überprüfung der Tauglichkeit der vorgestellten Controller reicht ein virtueller Ansatz aber aus. Die resultierende Klasse Fridge enthält zum einen eine Methode clock(), welche mittels Formel 3.1 aus Abschnitt 3.3 in jedem simulierten Takt die eigene Temperatur aktualisiert. Weiterhin enthält die Klasse die Methoden enableCooling() und disableCooling(), welche das An- und Abschalten der Kühlelektronik repräsentieren. Abschließend stellt diese Klasse dann mittels der Methode getTemperature() noch den benötigten Sensor zum Auslesen der aktuellen Temperatur zur Verfügung. Damit die Überprüfung der Controller komfortabel durchgeführt werden kann, wurde die bereits in Kapitel 3 vorgestellte Bibliothek *JFreeChart* auch hier zur Darstellung der resultierenden Temperatur- und Lastkurve des simulierten Kühlgerätes verwendet. Dazu enthält das virtuelle Kühlgerät noch zwei Zeitreihenobjekte, welche die Werteänderungen protokollieren und nach Ablauf der Simulation grafisch dargestellt und ausgewertet werden können.
Um die Startzustände der Basis-Controller wie bei den bisher verwendeten Simulationseinstellungen im Verhältnis 22% : 78% zu streuen musste zudem eine neue Transition erstellt werden, die ähnlich zur äußeren *null-transition* in Abbildung 7.2 den Automaten im Zustand warming starten lässt. Gemäß der Streuung wird dann pro Controller eine von beiden Start-Transitionen verwendet.

7.4.1 Simulation der Automaten

Im Folgenden werden nun die Simulationsergebnisse der Controller dargestellt und erläutert. Abbildung 7.10 zeigt kombiniert die Zeitreihen der Simulationen des Basis-Controllers (rot), der DSC-*load*-Steuerung ohne Dämpfung (blau) sowie mit zufallsbasierter Dämpfung (grün). Das Steuerungssignal traf in $t_{notify} = 1\,h\,30\,m$ ein. Der Streuungsparameter wurde auf spread $= 0\,min$ gesetzt, um die Schaltpunkte des Automaten deutlich zu machen. Die Grafik zeigt, dass alle drei simulierten Automaten ihre Funktion korrekt ausführen. Der BaseController produziert den bereits aus Abbildung 3.3 bekannten Verlauf eines einzelnen unmodifizierten Kühlgerätes.

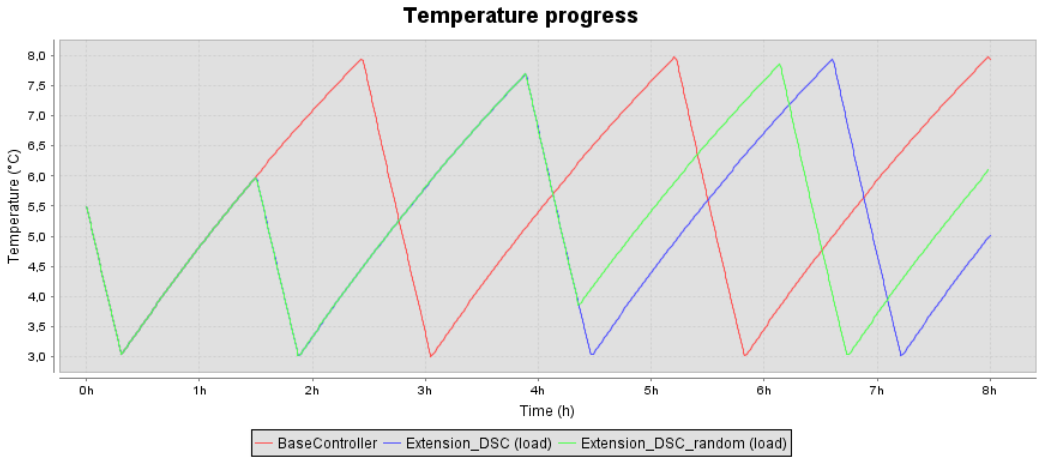


Abbildung 7.10: Temperaturverlaufssimulationen der FSM BaseController, Extension_DSC und Extension_DSC_random.

Nach 90 Minuten simulierter Zeit knicken die Temperaturkurven der Automaten Extension_DSC und Extension_DSC_random plötzlich ab, um dem empfangenen Signal Folge zu leisten und den thermischen Speicher zu leeren. Während die blaue Kurve des Automaten ohne Dämpfung fortan ihrem neuen Verlauf folgt, ändert die grüne Kurve des Automaten mit zufallsbasierter Dämpfung im Zeitpunkt $t_{rnd} = 4\,h\,21\,m$ erneut ihren Verlauf, was der zufälligen Streuung durch diese Schaltung entspricht. Somit haben alle drei Automaten ihre Aufgabe erfüllt. Simulationen mit variierenden Zeitpunkten und abweichendem spread bestätigen dies.
Auffällig an den Ergebnissen ist bei genauem Hinsehen jedoch der Zeitpunkt $t = 3\,h\,53\,m$, in welchem die modifizierten Geräte nach erfolgtem Steuerungssignal das erste Mal eine vollständige Aufwärmphase durchgeführt haben. Denn statt bis zu der Maximaltemperatur $T_{max} = 8.0°C$ zu warten, schalten sie bereits bei etwa $7.7°C$ in die Kühlphase. Dieses Verhalten konnte leider bei jeder durchgeführten Simulation festgestellt werden und liegt an der Annahme eines linearen Temperaturverlaufes. Wie bereits erwähnt werden die Werte $\tau_{cooling}$ und $\tau_{warming}$ nach jedem Phasenwechsel aktualisiert. Die Berechnung ist darauf ausgelegt, auch bei nicht vollständig durchlaufenen Phasen die Werte zu aktualisieren, und dabei den Wert auf Basis des Anteils der durchlaufenen Phase im Verhältnis zur maximalen Phasenlänge linear hochzurechnen. Die entsprechenden Formeln sind in Abbildung 7.2 in den exit-Aktionen der beiden Hauptzustände dargestellt. Da aber der Temperaturverlauf insbesondere der Aufwärmphase nicht linear, sondern gekrümmt verläuft, entsteht

hier ein Fehler in der Berechnung von $\tau_{warming}$. Dieser ist hier für die verkürzte nachfolgende Aufwärmphase verantwortlich, da auf seiner Basis unter anderem der Wert τ_{switch} berechnet wird (siehe Abbildung 7.2, entry-Aktion des Zustands warming). Um diesen Umstand noch deutlicher zu machen, ist in Abbildung 7.11 der entsprechende Zeitabschnitt vergrößert dargestellt und mit Hilfslinien versehen worden. Dieser Fehler tritt auch in den folgenden Simulationen auf und kann leider

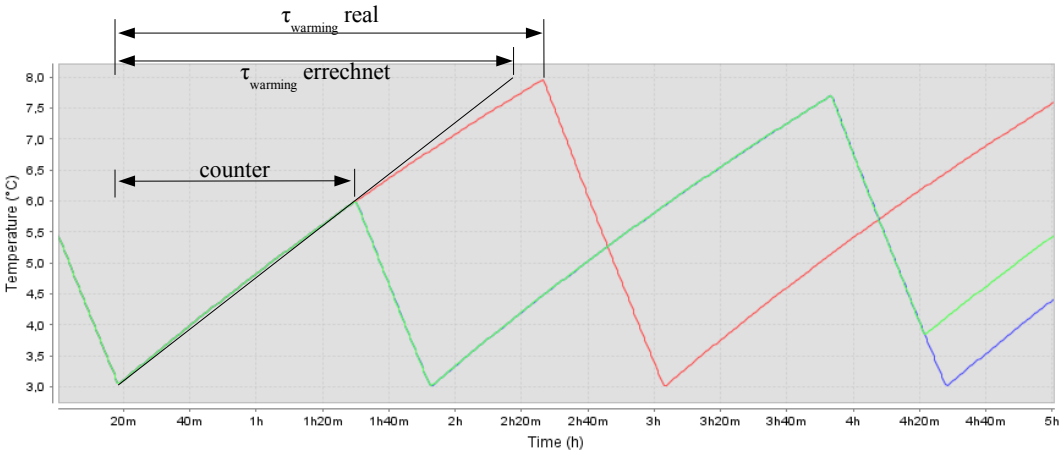


Abbildung 7.11: Fehler der linearen Berechnung von $\tau_{warming}$.

ohne Weiteres nicht korrigiert werden, dazu wäre die Kenntnis der exakten Geräteparameter sowie die anschließende Verwendung von Formel 3.4 zur Berechnung von $\tau_{warming}$ notwendig.

In Abbildung 7.12 ist die Arbeitsweise der Controller Extension_DSC_stateful (blau) sowie Extension_DSC_stateful_fullwidth (grün) dargestellt, und die rote Kurve zeigt wieder den unmodifizierten BaseController. Auch hier ist zu erkennen, dass die Controller zwar wie gewünscht dem Signal DSC-*load* Folge leisten und später zu ihren spezifischen Zeitpunkten die Zustandswiederherstellung durchführen. Allerdings wirkt sich auch hier der Fehler in der linearen Berechnung aus. Die blaue Kurve der Zustandswiederherstellung als *half-width*-Variante (vgl. Abbildung 5.3) erkennt korrekt das Zusammentreffen des modifizierten und des regulären Temperaturverlaufes in Zeitpunkt $t_{\gamma^h} = 2\,h\,45\,m$, verlässt aber in $t = 4\,h\,58\,m$ zu früh die folgende Aufwärmphase. Die Zustandswiederherstellung als *full-width*-Variante (vgl. Abbildung 5.4) sollte eigentlich das zweite Zusammentreffen mit der regulären Kurve in

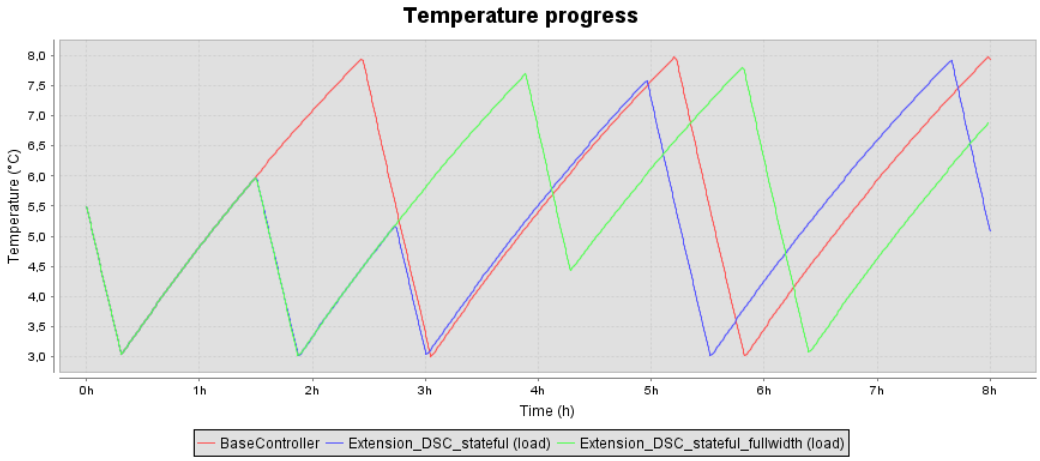


Abbildung 7.12: Temperaturverlaufssimulationen der FSM BaseController, Extension_DSC_stateful und Extension_DSC_stateful_fullwidth.

ca. $t_{\gamma f} = 4\,h\,10\,m$ errechnen, verpasst diesen Punkt jedoch aufgrund des fehlerhaften Wertes $\tau_{warming}$ und wechselt erst in $t' = 4\,h\,17\,m$ die Phase.
Diese Abweichungen vom gewünschten Verlauf deuten bereits darauf hin, dass die zustandsbasierten Dämpfungsmechanismen eher weniger geeignet für eine reale Umsetzung sind, da sie zu sehr von den errechneten Werten $\tau_{warming}$ und $\tau_{cooling}$ abhängen. Die Simulation des DSC-*unload*-Signals brachten keine zu den obigen Ergebnissen abweichenden Erkenntnisse und werden daher hier nicht weiter demonstriert. Im Folgenden werden nun die Controller für das TLR-Steuersignal simuliert und getestet. Abbildung 7.13 zeigt exemplarisch den Verlauf, der sich nach einem TLR-Signal mit $t_{notify} = 1\,h\,30\,min$, $\tau_{preload} = 30\,min$, und $\tau_{reduce} = 2\,h$ ergibt. Das Gerät wurde bei diesen Parametern in die Klasse C eingeordnet und befand sich dabei in der Aufwärmphase, daher gab es einen Schnittpunkt mit der Geraden s_{BA} und der Automat hat den Weg über den Zustand intersect_s_{BA} gewählt (vgl. Abbildung 7.6). Als Folge daraus wartet die Schaltung bis zum Zeitpunkt $\gamma_{s_{BA}} = 1\,h\,22\,m$, wo das Gerät in die Phase Kühlen versetzt wird und bis zum Ende des Intervalles $\tau_{preload}$ kühlt. Nach diesem Intervall hat es dann die errechnete Grenztemperatur $T_{max-act}$ erreicht, mit der es das Reduktionsintervall exakt mit Aufwärmen überstehen kann. Nach der Reduktion im Zeitpunkt $t_{notify} + \tau_{preload} + \tau_{reduce} = 180\,min$ sollte das Gerät also seine Maximaltemperatur T_{max} erreicht haben und wieder in den Kühlbetrieb schalten. Dass dieser Zeitpunkt nicht ganz eingehalten wird liegt wiederum an dem Fehler durch die lineare Annäherung von $\tau_{warming}$.

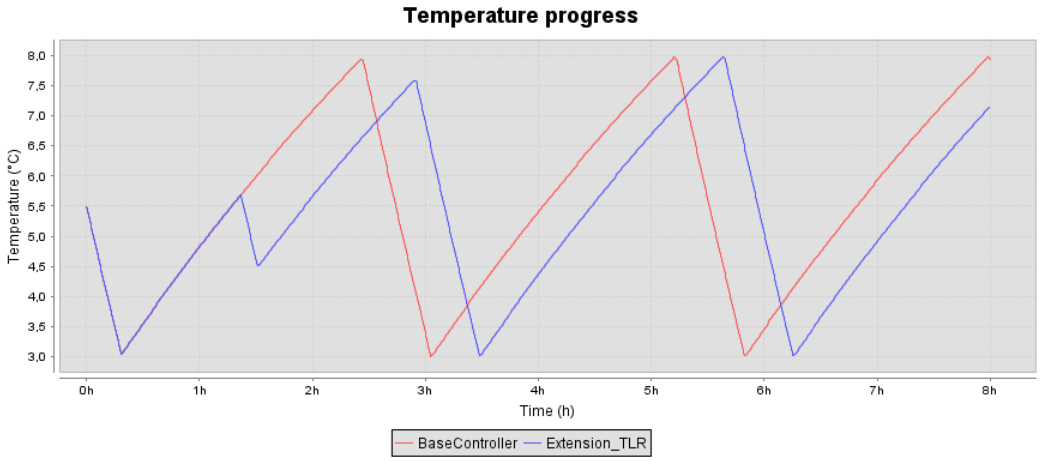


Abbildung 7.13: Temperaturverlaufssimulationen der FSM BaseController und Extension_TLR mit $t_{notify} = 1\,h\,00\,m$, $\tau_{preload} = 30\,min$, $\tau_{reduce} = 1\,h\,30\,m$.

Die Simulationen, in denen das Gerät in die verbleibenden Klassen eingeordnet wurde, sind in Anhang C enthalten. An dieser Stelle werden nun die Dämpfungsmechanismen getestet. Abbildung 7.14 zeigt daher die Simulationen der Controller TLR_extension_stateful (grün) und TLR_extension_random (gelb). Die Temperaturkur-

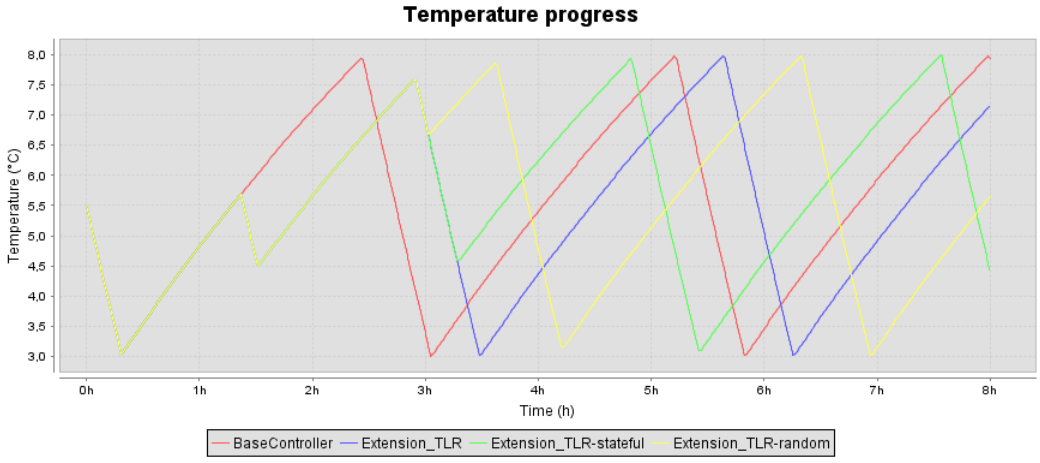


Abbildung 7.14: Temperaturverlaufssimulationen der FSM BaseController, Extension_TLR, TLR_extension_stateful sowie TLR_extension_random.

ven des Basis-Controllers sowie der ungedämpften TLR-Steuerung sind zum Zweck des Vergleiches ebenfalls enthalten. Es ergibt sich ein Bild ähnlich der Ergebnisse der

Simulationen der DSC-Steuerung mit Dämpfung. Die zustandsbehaftete Dämpfung (grüne Kurve) verwendet einen nicht ganz korrekten Wert $\tau_{warming}$ für die Berechnung der Zustandswiederherstellung und weicht daher ein Stück von der angepeilten roten Kurve ab, indem zu früh in die Kühlphase geschaltet wird. Die zufallsbasierte Dämpfung (gelbe Kurve) arbeitet hingegen korrekt, indem sie in der dem Reduktionsintervall folgenden Kühlphase irgendwann abknickt, im vorliegenden Fall wurde vom Controller die Zufallstemperatur $T_{rnd} \approx 6.7°C$ gewählt.
In Abschnitt 6.2 wurde bereits gezeigt, dass die Dämpfungsmechanismen robust gegenüber Variationen der Geräteparameter sind. Dies gilt auch für die Parameter der Steuerungssignale, da die Dämpfungen erst nach erfolgter Steuerung agieren. Daher müssen hier keine weiteren Simulationen mit abweichenden Parametern durchgeführt werden.

Es konnte also mittels der Implementierung der Modelle in ausführbaren Java-Code gezeigt werden, dass die entwickelten Automaten valide sind und sich bis auf die Abweichungen durch den unvermeidbaren Fehler in der linearen Annäherung von $\tau_{warming}$ wie erwartet verhalten.

7.4.2 Auswirkungen des Linearisierungsfehlers

Um abschätzen zu können, wie sich der Fehler in der linearen Annäherung von $\tau_{warming}$ im realen Betrieb bei einer großen Menge von Controllern verhält, wurde das oben vorgestellte Framework um die Möglichkeit erweitert, mehrere Instanzen der Controller zu simulieren. Dazu musste unter anderem die Ebene der Datensammlung erweitert werden, um ähnlich wie bei den ereignisbasierten Simulationen im Mittelteil dieser Arbeit die aus einem Simulationslauf resultierenden Zeitreihen als Mittelwertkurve darstellen zu können. Konkret bedeutet dies, dass die Fridge Objekte nicht mehr selbst für die Haltung ihrer Zeitreihen verantwortlich sind, sondern dass sie in jedem Takt ihre aktuellen Werte einer zentralen Datenhaltungsinstanz mitteilen, welche dann den Mittelwert aller in diesem Takt erhaltenen Werte bildet und diesen als Datenpunkt in der resultierenden Zeitreihe speichert. Dies geschieht getrennt sowohl für die Temperatur- als auch für die Lastwerte.
Die folgenden Simulationen wurden jeweils mit einer Populationsgröße von 5000 Einheiten durchgeführt, wobei eine Einheit jeweils aus virtuellem Kühlgerät, Basis-Controller sowie optionalen Controller-Erweiterungen besteht. Die Geräteparameter wurden ausgehend von den Erkenntnissen aus Abschnitt 6.2 über die Population

gestreut, wobei die Starttemperatur der Geräte gleichverteilt, und die restlichen Parameter normalverteilt sind. Wie bei allen bisherigen Simulationen mit großer Population begannen 22 % der Geräte die Simulation mit der Phase Kühlen. Ein wichtiger Unterschied zu den bisherigen Simulationen ist jedoch die auf vier bis sechs Stunden verlängerte Initialisierungsphase $\tau_{init} = [0.0, t_{notify}]$, bevor ein Signal abgesetzt wird. Diese ist notwendig, damit alle Controller mindestens einen vollständigen Zyklus $\tau_{cycle} = \tau_{cooling} + \tau_{warming}$ durchlaufen und diese Richtwerte entsprechend ihren Geräteparametern anpassen können. Außerdem wurde die Simulationslänge insgesamt verlängert, um nicht nur den unmittelbaren Bereich um die abgesetzten Steuersignale darzustellen, sondern auch den darauf folgenden Zeitraum, in welchem sich unter Umständen Folgeeffekte zeigen können. Um eventuelle Unterschiede zu den Simulationen der ereignisbasierten Modelle genauer zeigen und analysieren zu können, werden die folgenden Simulationen je einmal mit dem ereignisbasierten Modell (in den Diagrammen bezeichnet mit „SimKit“) und einmal mit dem Controller-Simulator (Bezeichnung „FSM“) durchgeführt, und die resultierenden Kurven dann miteinander verglichen. Dadurch sollte sichtbar werden, inwiefern sich der Linearisierungsfehler bei einer großen Gerätepopulation auswirkt. Die Diagramme weichen in der Darstellung etwas von den bisherigen ab, da sie mit einer Tabellenkalkulation erstellt wurden, um weitergehende Analysen durchführen zu können.

Abbildung 7.15 zeigt die Ergebnisse der Simulation mit DSC-*load*-Signal ($t_{notify} = 240\,min$, spread $= 10\,min$) und zustandsbasierter Dämpfung (*half-width*-Variante, vgl. Abbildung 5.3). Die Lastkurve der FSM-Simulation zeigt gegenüber ihrem Pendant eine leichte Schwingung im Anschluss an die dem Signal folgende Aufwärmphase. Das Maximum dieser Abweichung ist bei ca. $t = 430\,min$ erreicht, liegt aber nur etwa $3\,W$ über dem globalen Mittel von $\bar{Q} \approx 15.5\,W$. Ein Vergleich der von den Zeitreihen aufgespannten Flächen hat ergeben, dass sich die Lastkurve der SimKit-Simulation über $0.2325\,kWh$ erstreckt, während die FSM-Simulation $0.2339\,kWh$ überspannt. Dies entspricht einer Abweichung von 0.59 %. Die Temperaturkurven unterscheiden sich in ihrem Integral um 1.45 %, wobei auch hier die FSM-Simulation etwas mehr Fläche überspannt.

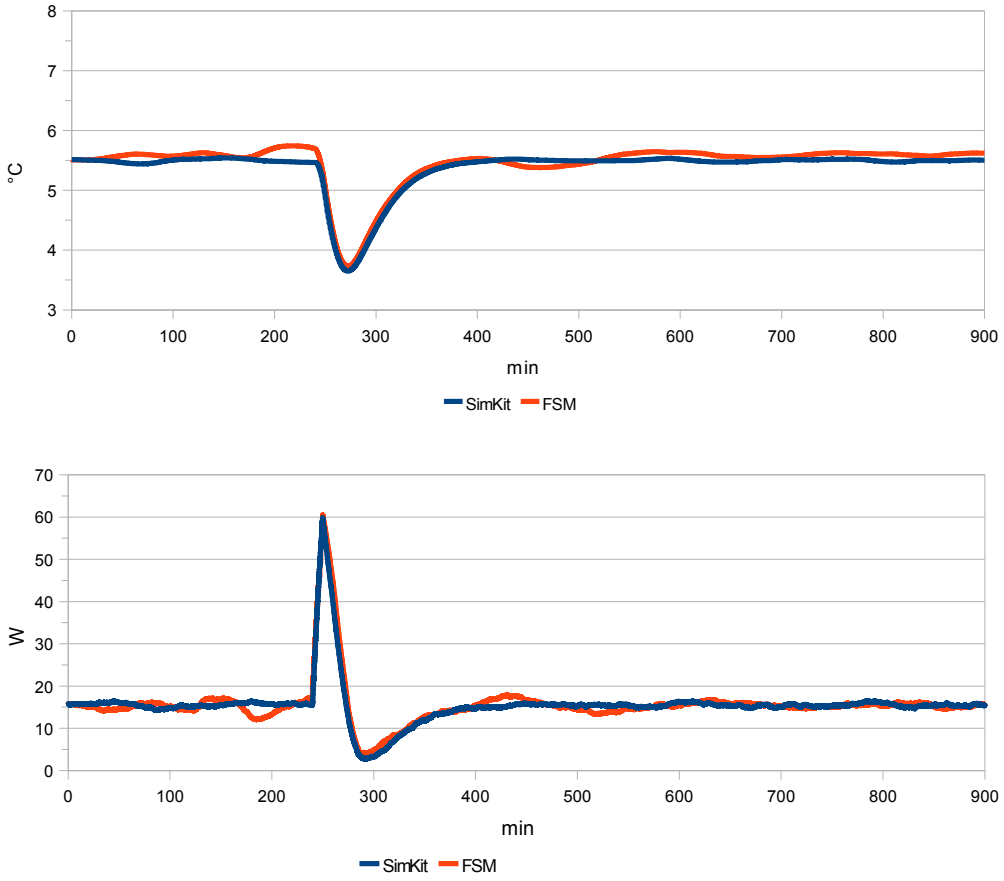


Abbildung 7.15: Vergleich der SimKit- und FSM-Simulation mit DSC-*load*-Signal und zustandsbasierter Dämpfung (*half-width*).

Abbildung 7.16 zeigt die Ergebnisse einer ähnlichen Simulation mit DSC-*load*-Signal ($t_{notify} = 240\,min$, spread $= 10\,min$) und zustandsbasierter Dämpfung (*full-width*-Variante, vgl. Abbildung 5.4). Ähnlich wie im vorherigen Fall zeigt auch hier

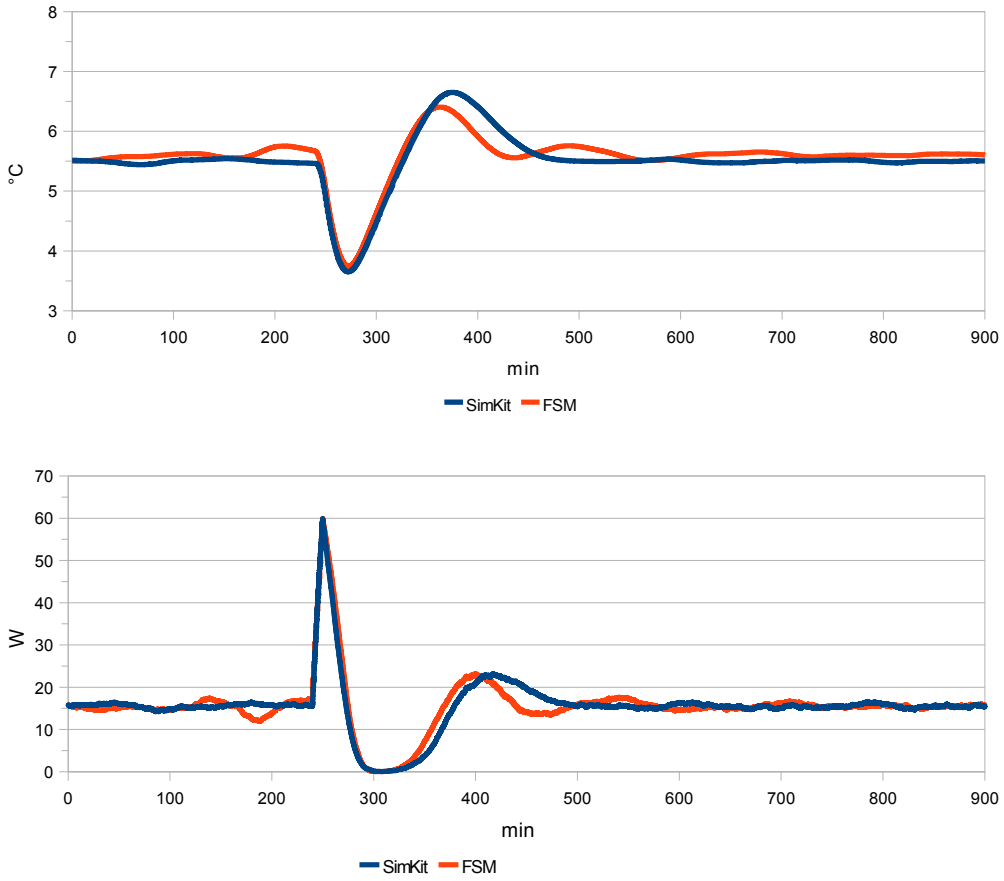


Abbildung 7.16: Vergleich der SimKit- und FSM-Simulation mit DSC-*load*-Signal und zustandsbasierter Dämpfung (*full-width*).

die Lastkurve der FSM-Simulation eine leichte Schwingung nach der Signalphase. Zu erkennen ist dies in etwa $t = 460\,min$, wo die Kurve etwas unter das globale Mittel fällt, um kurz darauf etwas über das globale Mittel anzusteigen, bevor sie sich dem Mittel nähert. Ein Flächenvergleich ergab bei diesen Lastkurven, dass die FSM-Kurve mit $0.2328\,kWh$ wiederum 0.13% mehr Fläche überspannt als die SimKit-Kurve. Auch die Abweichungen bei den Temperaturkurven ähneln den vorherigen Ergebnissen, hier weicht das Integral der FSM-Kurve um 1.2% in positive

Richtung gegenüber der Fläche der SimKit-Kurve ab.
In Abbildung 7.16 sind anschließend die Ergebnisse einer Simulation mit DSC-*load*-Signal ($t_{notify} = 240\,min$, spread $= 10\,min$) und zufallsbasierter Dämpfung zu sehen. Hier zeigt die Lastkurve der FSM-Simulation gegenüber der SimKit-Simulation ledig-

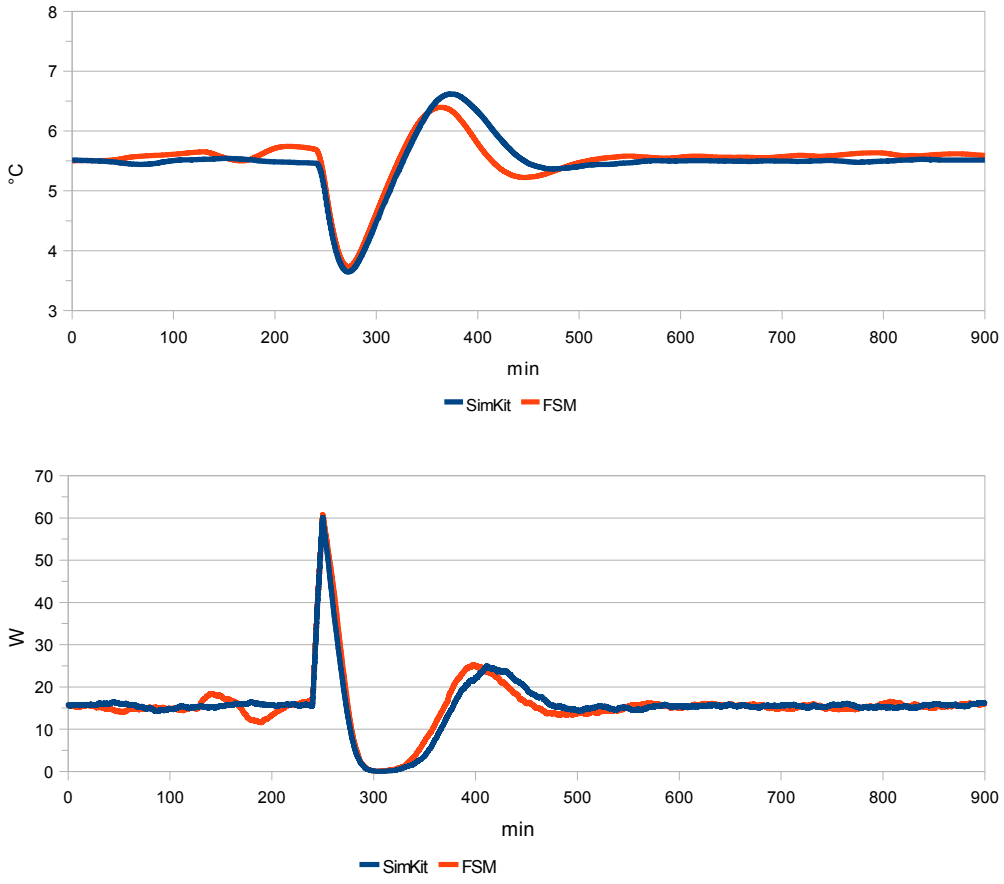


Abbildung 7.17: Vergleich der SimKit- und FSM-Simulation mit DSC-*load*-Signal und zufallsbasierter Dämpfung.

lich eine in ca. $t = 330\,min$ etwas eher beginnende Erhöhung der Last nach erfolgtem Steuersignal, gefolgt von einer leichten Absenkung der Last unter den globalen Mittelwert im Zuge der Desynchronisation in etwa $t = 480\,min$. Der Flächenvergleich ergab hier ähnlich zu den vorhergehenden Simulationen, dass die FSM-Kurve mit $0.2333\,kWh$ etwas mehr Fläche überspannt als die SimKit-Kurve mit $0.2326\,kWh$, dies entspricht einer Abweichung von $0.26\,\%$. Die Temperaturkurven unterscheiden

sich etwas weniger als die vorhergehenden, die Abweichung im Integral der FSM-Kurve beträgt hier 0.71 % gegenüber der SimKit-Kurve.

Die Erklärungen für die vorliegenden Abweichungen sind je nach Dämpfungsart unterschiedlich. Für die zuletzt beschriebene Lastkurve des FSM-Controllers mit zufallsbasierter Dämpfung erklärt sich das Verhalten durch die bereits aus Abbildung 7.10 bekannte verkürzte Aufwärmphase nach dem erfolgten Steuersignal, da der Automat Extension_DSC_random den Zeitpunkt der Dämpfung auf Basis der vor dem Signal noch korrekten Zeiten $\tau_{warming}$ und $\tau_{warming}$ errechnet. Durch die verkürzte Aufwärmphase jedoch schalten die Controller alle etwas eher in die darauf folgende Kühlphase, sodass zum Zeitpunkt der Dämpfung potenziell insgesamt mehr gekühlt wurde als erwartet. Anders ausgedrückt wurde der optimale Zeitpunkt der Dämpfung verpasst und die Controller befanden sich etwas länger in Synchronisation als errechnet. Die Erklärung des Verhaltens der zustandsbasierten Dämpfungen ist komplizierter, hier wird vermutlich ebenfalls ein Zusammenhang zwischen der verkürzten Aufwärmphase und dem Zeitpunkt der Wiederherstellung bestehen. Zur Klärung wurden einige Einzelsimulationen durchgeführt, Abbildung 7.18 zeigt den einen solchen Testlauf. In dem Diagramm sind die Temperaturkurven von

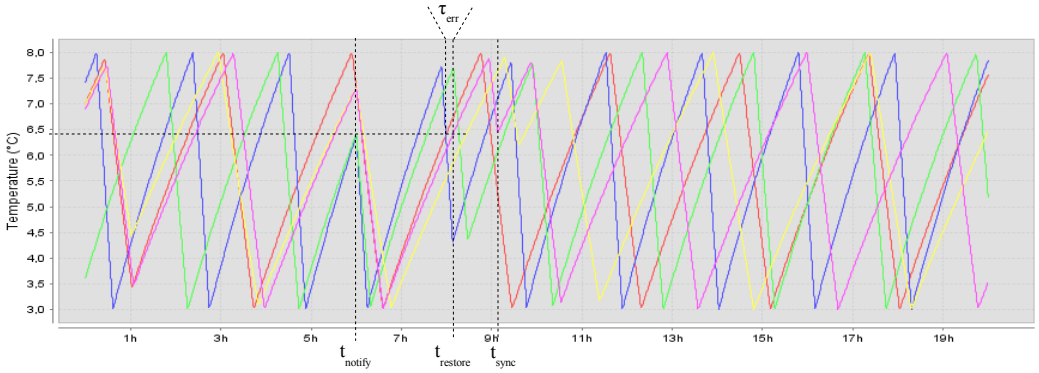


Abbildung 7.18: Temperaturverlaufssimulationen von einzelnen Controllern mit Zustandswiederherstellung.

fünf simulierten Einheiten mit Zustandswiederherstellung (*full-width*, siehe Abbildung 5.4) dargestellt. Die Geräteparameter wurden gestreut, das DSC-Signal wurde in $t_{notify} = 6\,h\,00\,m$ abgesetzt. Betrachtet man nun den Zeitpunkt des Signales, so ist zu sehen, dass sich bis auf die rote Kurve alle Controller in der Phase Aufwärmen

befanden und durch das Signal in die Phase Kühlen schalten, um ihre thermischen Speicher zu füllen. Nach der Absenkung der Temperatur auf T_{min} und der darauf folgenden Aufwärmphase bis T_{max} sollte nun eigentlich die Zustandswiederherstellung beginnen. Aufgrund der verkürzten Aufwärmphase verpassen die Controller jedoch die Temperatur, die sie zum Zeitpunkt des Signales hatten und kühlen noch ein Stück weiter, bevor die Phase gewechselt wird. Dies ist deutlich an der blauen Kurve zu erkennen: in t_{notify} befand sich das Gerät in der Phase Aufwärmen und hatte eine Temperatur von etwa $T_{notify} \approx 6.4°C$. Rechnerisch müsste es diese Temperatur durch das Steuersignal erneut im Zeitpunkt $t_{restore} \approx 8\,h\,10\,m$ erreichen, daher wird zu diesem Zeitpunkt der Zustand wiederhergestellt und erneut in die Phase Aufwärmen geschaltet. Die fehlerhafte Annäherung von $\tau_{warming}$ bewirkt hier jedoch, dass das Gerät die Zieltemperatur $T_{restore} = T_{notify}$ schon etwas eher erreicht als errechnet, und die Zustandswiederherstellung daher um τ_{err} Minuten zu spät statt findet. Der wichtige Punkt ist nun, dass die Verzögerung der Wiederherstellung desto größer wird, je geringer die Temperatur T_{notify} zum Zeitpunkt des Signales war (vgl. Abbildung 7.11). Vergleicht man daher den Verlauf der blauen Kurve beispielsweise mit dem der pinken Kurve, welche in t_{notify} eine höhere Temperatur hatte und daher eine geringere Verzögerung der Zustandswiederherstellung hat, so fällt auf, dass diese beiden Kurven nach erfolgter Zustandswiederherstellung in $t_{sync} = 9\,h\,10\,m$ plötzlich nahezu synchronisiert sind, obwohl sie vor erfolgtem Steuersignal weit voneinander entfernt waren. Die grüne und gelbe Kurve zeigen ein ähnliches Verhalten. Die Vergrößerung des linearen Fehlers bei niedrigerer Temperatur in t_{notify} bewirkt also eine derartige Verzögerung der Zustandswiederherstellung, dass die Geräte für kurze Zeit mit solchen Geräten synchronisiert werden, welche eine höherere Temperatur in t_{notify} hatten und deren Verzögerung der Zustandswiederherstellung geringer war. Die Synchronisation wird durch die gestreuten Geräteparameter kurz darauf wieder aufgehoben.

Da die Simulationsergebnisse für das DSC-*unload*-Signal insbesondere bei den Lastkurven keine auffälligen strukturellen Unterschiede zeigten, werden die entsprechenden Diagramme hier nicht dargestellt, sind aber in Anhang D einzusehen. Die Abweichungen der Integrale der FSM-Kurven gegenüber den SimKit-Kurven sind jedoch der Vollständigkeit halber in Tabelle 7.1 abgebildet.

Dämpfung	Abweichung
zustandsbasiert, *half-width*, Temperatur	1.77 %
zustandsbasiert, *half-width*, Last	−0.31 %
zustandsbasiert, *full-width*, Temperatur	1.69 %
zustandsbasiert, *full-width*, Last	−0.12 %
zufallsbasiert, Temperatur	2.26 %
zufallsbasiert, Last	−0.2 %

Tabelle 7.1: Integralabweichungen der FSM-Kurven gegenüber den SimKit-Kurven mit DSC-*unload*-Signal.

Abbildung 7.19 zeigt nun die Ergebnisse einer Simulation mit TLR-Signal ($t_{notify} = 360$, $\tau_{preload} = 30\,min$, $\tau_{reduce} = 90\,min$) und zustandsbasierter Dämpfung. Positiv zu bemerken ist zunächst, dass die Berechnung des Verhaltens im Vorlaufzeitraum $\tau_{preload}$ offenbar korrekt funktioniert, da die Population insgesamt ihre thermischen Speicher in dieser Zeit füllt, um nachfolgend im Reduktionsintervall wie gefordert die Last vollständig abzusenken. Der Anstieg zum Ende des Intervalles ist durch die Parameterstreuung bedingt und war auch schon in Abbildung 6.5 zu beobachten, allerdings steigt die Lastkurve der FSM-Simulation hier schon etwas eher an als die Kurve der SimKit-Simulation. Weitere Unterschiede zeigen sich bei der folgenden Desynchronisation der Controller. Überraschend ist hier, dass die Zustandswiederherstellung trotz der linearisierten Berechnungen nahezu ohne Abweichung arbeitet, es sind lediglich eine leichte Überkühlung sowie nachfolgend geringe Oszillationen mit einer anfänglichen Abweichung von $\pm 3\,W$ zum globalen Mittel zu erkennen. Der Vergleich der Kurvenintegrale ergab, dass die Lastkurve der FSM-Simulation um −0.91 % von der Lastkurve der SimKit-Simulation abweicht, also insgesamt etwas weniger Fläche überspannt. Die FSM-Temperaturkurve hingegen weicht um 1.31 % ab, liegt also etwas über der Kurve der SimKit-Simulation.

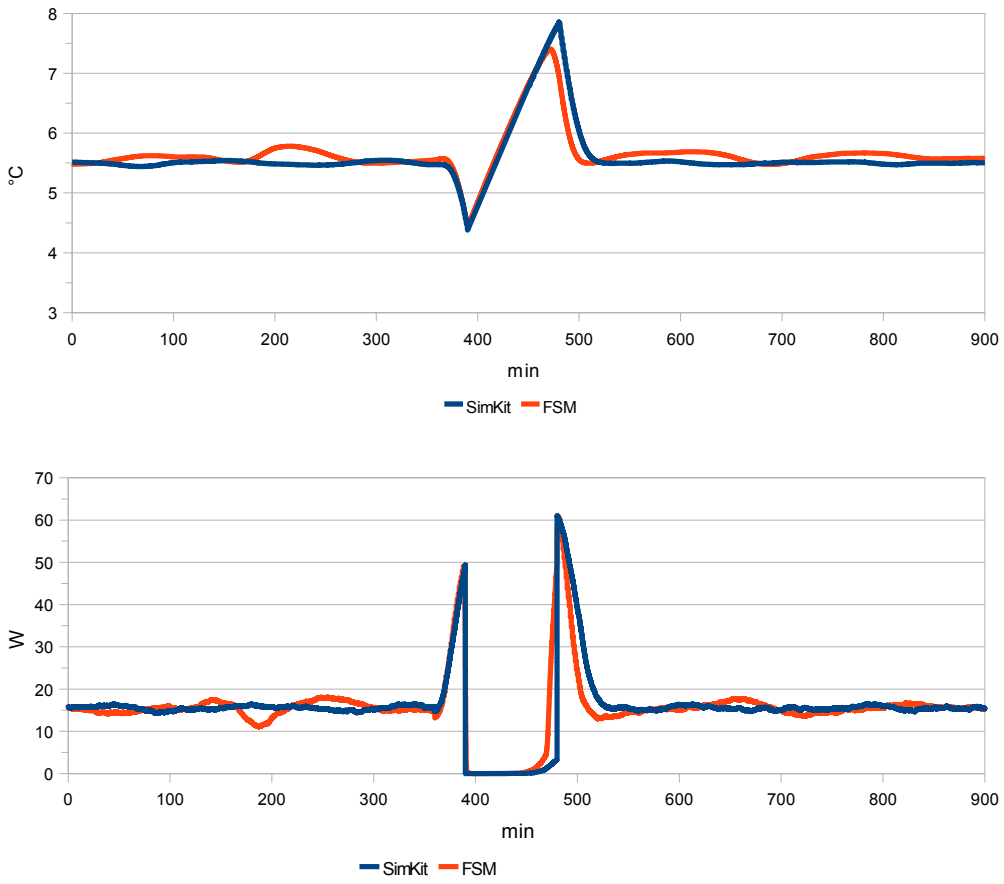


Abbildung 7.19: Vergleich der SimKit- und FSM-Simulation mit TLR-Signal und zustandsbasierter Dämpfung.

Abbildung 7.20 zeigt abschließend die Ergebnisse einer Simulation mit TLR-Signal ($t_{notify} = 360$, $\tau_{preload} = 30\,min$, $\tau_{reduce} = 90\,min$) und zufallsbasierter Dämpfung. Die Kurven zeigen ein sehr ähnliches Bild wie bei obigen Simulationsergebnissen

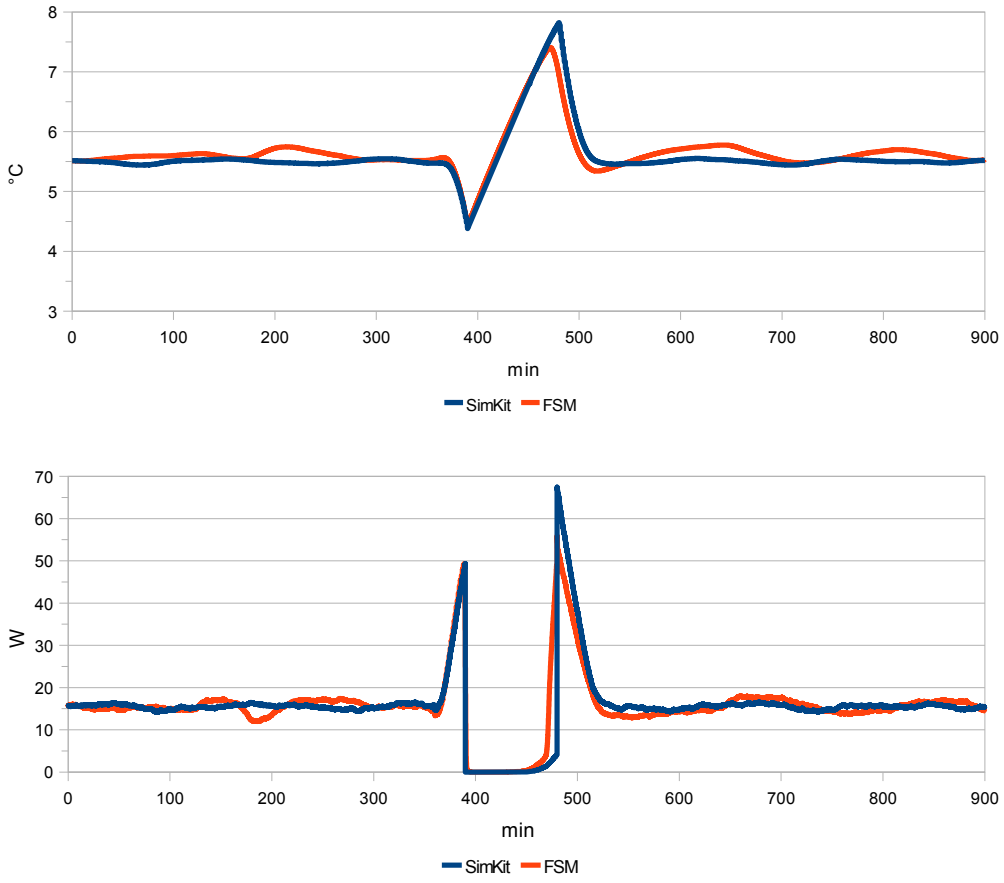


Abbildung 7.20: Vergleich der SimKit- und FSM-Simulation mit TLR-Signal und zufallsbasierter Dämpfung.

der zustandsbasierten Dämpfung mit nahezu identischen Abweichungen. Der Flächenvergleich zeigte hier, dass die Lastkurve der FSM-Simulation um $-0.02\,\%$ von der SimKit-Lastkurve abweicht, bei der Temperaturkurve beträgt die Abweichung $1.22\,\%$.
Die Abweichungen der TLR-Simulationen werden vermutlich wie bei der DSC-*load*-Variante durch verkürzte Aufwärmphasen verursacht.

Insgesamt scheinen die entwickelten Automaten bei einer großen Population von Controllern und virtuellen Kühlgeräten mit gestreuten Geräteparametern trotz des signifikanten Fehlers in der linearen Annäherung von $\tau_{warming}$ global gesehen korrekt zu arbeiten. Die prozentualen Abweichungen der von den Kurven aufgespannten Flächen beträgt maximal 2.26 % im Vergleich zu äquivalenten Kurven der SimKit-Simulationen. Die Berechnung der Integrale muss hier allerdings als Hilfsmittel betrachtet werden und darf nicht allein gesehen interpretiert werden, da die hieraus errechneten Abweichungen keine strukturellen Verlaufsunterschiede berücksichtigen. So würden etwa eine konstant bei einem Wert p horizontal verlaufende Kurve und eine gleichmäßig um den Wert p schwingende Kurve die gleiche Fläche aufspannen, ihr struktureller Verlauf hingegen unterscheidet sich wesentlich. Daher muss zusätzlich auf die lokalen Gegebenheiten der Verläufe (in Bezug auf Signalzeitpunkte etc.) geachtet werden. Aber auch diese Betrachtungen zeigten keine derartigen Abweichungen im Kurvenverlauf, wie es nach der Darlegung des Ausmaßes des Linearisierungsfehlers (Abbildung 7.11) vielleicht zu erwarten gewesen wäre.

7.5 Zusammenfassung

Die in diesem Kapitel vorgestellten Zustandsmodelle der einzelnen Controllerkomponenten zeigen, dass der modulare Aufbau von weitestgehend unabhängigen Schaltungen nicht schwieriger zu entwickeln ist als ein einzelnes, integriertes Modell. Der Zustandsgraph aus Abbildung 7.2 beschreibt dabei die grundlegende Funktionalität des Controllers, um den normalen Betrieb des Kühlgerätes zu gewährleisten. Dieses Modell kann dann durch ein oder auch mehrere Module in seiner Funktion erweitert werden, ohne die Basiskomponente dazu anpassen zu müssen. Dafür wurden die Automatenmodelle aus Abbildung 7.3 und Abbildung 7.6 vorgestellt, die die Realisierung der DSC- und der TLR-Steuerung darstellen und lediglich lesenden Zugriff auf einige Parameter der Basiskomponente sowie die Möglichkeit, Signale an diese Komponente schicken zu können, zum Betrieb benötigen. Um darauf nun etwa einen der in dieser Arbeit entwickelten Desychronisationsmechanismen zu realisieren, reicht es zumindest für die TLR-Steuerung aus, eines der beiden Module aus Abbildung 7.7 und Abbildung 7.8 hinzuzufügen. Die Voraussetzungen sind die gleichen: lesender Zugriff auf Geräteparameter sowie die Fähigkeit, Steuersignale an den Basiscontroller zu schicken. Im Falle der DSC-Steuerung wurde wie im entsprechenden Absatz erläutert auf eine modular basierte Realisierung der Dämpfungsmechanismen verzichtet, hier muss stattdessen die DSC-Komponente durch eine entsprechend er-

weiterte Version ersetzt werden.

Betrachtet man die Anforderungen an die Hardware der einzelnen Module, so ergibt sich ein leicht inhomogenes Bild. Zunächst einmal benötigen alle Module eine ALU, um die zum Betrieb erforderlichen Berechnungen durchzuführen, sowie eine Echtzeituhr zur Berechnung und Ausführung der Schaltpunkte für die einzelnen Zustandsübergänge. Der Basiscontroller erfordert weiterhin Zugriff auf den Sensor, der die aktuelle Innentemperatur des Gerätes ausliest, sowie auf die Werte T_{min} und T_{max} (können vordefiniert und fest programmiert sein). Außerdem wird ein Speicher mit Schreibzugriff benötigt, in dem die Werte T_{from}, T_{dest}, counter, τ_{switch}, $\tau_{cooling}$ und $\tau_{warming}$ abgelegt werden. Zusätzlich muss der Controller für Erweiterungskomponenten lesenden Zugriff auf diesen Speicher bieten, da mit diesen Informationen das Verhalten des Gerätes komplett beschrieben werden kann, was für die Berechnungen der Erweiterungsmodule notwendig ist. Optional kann das Basismodul auch die Funktionen $a_c()$, $a_w()$, $\tau_{reqc}(T_{from}, T_{dest})$ und $\tau_{reqw}(T_{from}, T_{dest})$ zur Verfügung stellen, damit diese nicht von den Erweiterungen erneut implementiert werden müssen. Dies ist aber nicht zwangsweise notwendig, der Zugriff auf den Speicher mit den aktuellen Parametern reicht hierfür aus. Abschließend benötigt das Basismodul die Fähigkeit, die Steuersignale ev_cooling(T) und ev_warming(T) von Erweiterungsmodulen empfangen zu können.
Die Erweiterungsmodule DSC und TLR benötigen im Speziellen eine Kommunkationsschnittstelle, über die die extern gesendeten Steuersignale ev_load(spread) und ev_unload(spread) bzw. ev_reduce($\tau_{preload}, \tau_{reduce}$) empfangen werden können. Für das DSC-Modul ist außerdem ein Zufallszahlengenerator erforderlich, um aus dem übergebenen Parameter spread eine zufällige Verzögerung zu errechnen, sowie ein kleiner Zwischenspeicher, um die bool'sche Variable doUnload abzulegen (zur Vermeidung dieses Speichers könnte diese Variable aber auch durch zusätzliche Zustände modelliert werden). Das TLR-Modul ist hier komplexer und erfordert weitaus mehr arithmetische Operationen und Speicher, um die einzelnen Variablen der Berechnungen abzulegen.
Die Dämpfungsmechanismen haben für die DSC-Steuerung keine zusätzlichen Anforderungen, da im Falle der Zustandswiederherstellung lediglich eine Verzögerung berechnet und ein Signal abgesetzt werden muss, und in der zufallsbasierten Variante zusätzlich noch ein Zufallswert mit dem bereits vorhandenen Zufallszahlengenerator errechnet wird. Für die TLR-Steuerung ergibt sich hier jedoch eine andere Situation. Hier sind die Dämpfungen als eigenständige Module entworfen und haben daher

auch eigene Anforderungen. Das Modul zur Zustandswiederherstellung ist ähnlich komplex wie die TLR-Steuerung selbst und erfordert ebenfalls viele arithmetische Berechnungen sowie Speicher über verschiedene Zustände hinweg. Das zufallsbasierte Modul hingegen ist wesentlich einfacher aufgebaut, erfordert aber dafür einen Zufallszahlengenerator.

Insgesamt lässt sich festhalten, dass für das DSC-Signal beide Formen der Desynchronisation applikabel sind, da sie keine weiteren Anforderungen mitbringen, die der DSC-Automat nicht ohnehin schon hätte. Auch für die TLR-Steuerung sind die Unterschiede in der Anforderung geringer als erwartet. Die Zustandswiederherstellung erfordert relativ viel Rechenaufwand und außerdem beschreibbaren Speicher, dafür muss für die zufallsbasierte Variante wie bereits erwähnt ein Zufallszahlengenerator implementiert werden, welcher unter Umständen ebenfalls beschreibbaren Speicher benötigt. Dennoch würde ich an dieser Stelle definitiv die zufallsbasierte Methode empfehlen, da sie nicht so stark durch den Fehler in der linearen Annäherung von $\tau_{warming}$ in ihrer Funktion beeinträchtigt wird, wie es bei der Zustandswiederherstellung der Fall ist. Zudem ist das entsprechende Modul sehr übersichtlich gehalten und damit weniger fehleranfällig. Und letztlich stellt die Berechnung von Pseudozufallszahlen in eingebetter Hardware auch kein größeres Problem mehr dar (siehe dazu beispielsweise [18]).

Nachdem nun im Laufe dieser Arbeit verschiedene Steuersignale vorgestellt, zwei verschiedene Desynchronisationsstrategien entwickelt und die einzelnen Modelle aus der ereignisbasierten und der zustandsbasierten Sicht betrachtet wurden, stellt sich die Frage der praktischen Anwendbarkeit. In diesem Kapitel wurde deutlich, dass die Umsetzung eines Controllers mit diversen Erweiterungen keine große Hürde darstellt, hier würden sich beispielsweise programmierbare integrierte Schaltkreise (*field programmable gate arrays*, FPGAs) anbieten. Diese Bausteine können nach der Herstellung mit anwendungsspezifischer Logik programmiert werden, und einige Varianten enthalten auch eigene integrierte Speicherzellen oder eine als sogenannte *Hard-Core* realisierte Kommunikationsschnittstelle. Über einen externen Referenztakt kann FPGAs unkompliziert eine Echtzeituhr zur Verfügung gestellt werden, dieser Takt kann preisgünstig etwa durch einen Schwingquarz erzeugt werden. Statt FPGAs sind aber auch sogenannte anwendungsspezifische integrierte Schaltungen (*application specific integrated circuits*, ASICs) verwendbar. Diese Bausteine werden direkt mit der gewünschten Funktionalität und Logik hergestellt, können also im

Nachhinein nicht mehr verändert werden. Sie sind allerdings in großen Stückzahlen günstiger und arbeiten außerdem durch ihr auf die jeweilige Anwendung hin optimiertes Design effizienter als FPGAs.
Zur Anwendbarkeit zählt aber neben der konkreten Umsetzung in die Hardware auch der Umfang an Möglichkeiten der vorgestellten Controller. Daher wird im nächsten Kapitel das bisher identifizierte Steuerungspotenzial auf weitere denkbare Möglichkeiten hin untersucht.

Kapitel 8

Weiterführende Gedanken

Die Motivation zum grundlegenden Thema dieser Arbeit war die Möglichkeit, das Lastverschiebungspotenzial von Kühlgeräten in Privathaushalten auszunutzen, um ineffiziente, langwierige und teure Ausgleichsoperationen im Stromnetz durch Hilfskraftwerke zu vermeiden. Insbesondere die Simulationsergebnisse der TLR-Steuerung mit nachgeschalteter Dämpfung (siehe etwa Abbildung 5.10) zeigen, dass es möglich ist, über einen gewünschten Zeitraum die durch die Kühlgeräte verursachte Last auf ein Minimum zu reduzieren. Eine Nebenwirkung ist jedoch die vor- und die nachlaufende Lastspitze, die durch solch einen Eingriff entsteht. Um das Lastverschiebungspotenzial optimal auszunutzen wäre eine Steuerungsstrategie wünschenswert, die in einer großen Population von Kühlgeräten im kumulierten Mittel eine weitestgehend beliebig gestaltete Lastkurve produzieren kann. Im Folgenden soll daher untersucht werden, wie flexibel die beiden Steuerungssignale sind und was für Möglichkeiten sich hier bieten. Die in diesem Kapitel enthaltenen Simulationen wurden mit dem ereignisbasierten Simulationsframework aus dem Hauptteil dieser Arbeit erstellt. Das System zur Simulation von endlichen Automaten aus Abschnitt 7.4 würde hier zwar exaktere Ergebnisse liefern, benötigt aufgrund seiner detailgetreuen Modellierung allerdings auch ein Vielfaches der Zeit. Die für die folgenden Simulationen verwendeten Parameter sind bereits aus den vorhergehenden Kapiteln bekannt, es wurde jeweils die Starttemperatur der Geräte über die Population gleichverteilt, während die restlichen Geräteparameter jeweils normalverteilt gestreut wurden. Diese Konfiguration und damit alle folgenden Simulationen stellen also nur ein einzelnes Szenario dar und dürfen nicht allgemeingültig interpretiert werden. Es soll daher hier lediglich eine Basis für weiterführende Untersuchungen geschaffen werden, indem dieses exemplarische Szenario auf seine spezifischen Ausprägungen hin analysiert wird. Weiterhin soll

hier darauf hingewiesen werden, dass die im Folgenden untersuchten Ausprägungen der Signalparameter nicht alle für eine sinnvolle Lastreduktion geeignet sind. Dieses Kapitel untersucht jedoch die prinzipiell möglichen Kurvenformen, wie sie mit den vorhandenen Signaltypen erzeugt werden können, daher werden hier auch für den eigentlichen Zweck der Lastreduktion ungeeignete Parameter betrachtet. Bevor nun die Untersuchungen durchgeführt werden, muss zunächst spezifiziert werden, welche Kennwerte der resultierenden Kurven untersucht werden sollen. Daher zeigen die folgenden Grafiken exemplarische Lastkurven für das DSC-*load*- (Abbildung 8.1), das DSC-*unload*- (Abbildung 8.2) sowie für das TLR-Signal (Abbildung 8.3) inklusive der in den folgenden Abschnitten verwendeten Bezeichnern für die Kennwerte der einzelnen Kurven.

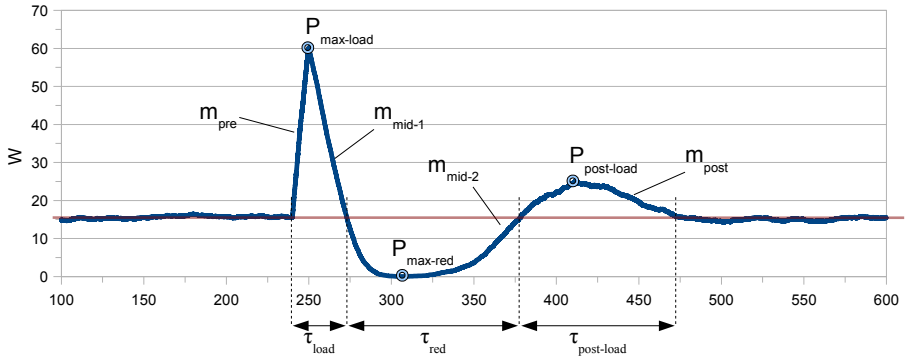


Abbildung 8.1: Kennwerte einer DSC-*load*-Lastkurve.

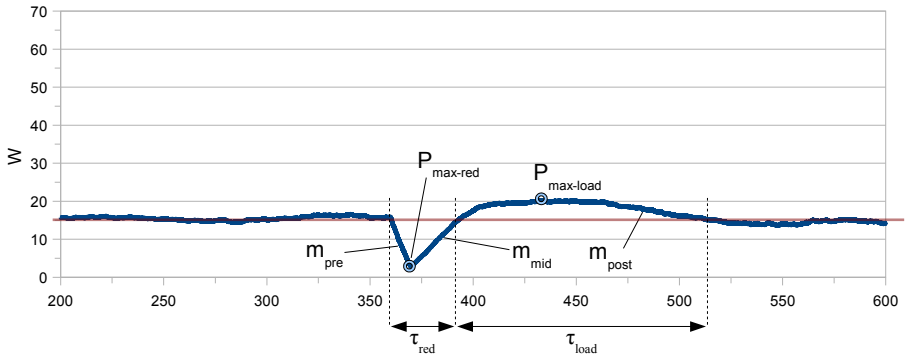


Abbildung 8.2: Kennwerte einer DSC-*unload*-Lastkurve.

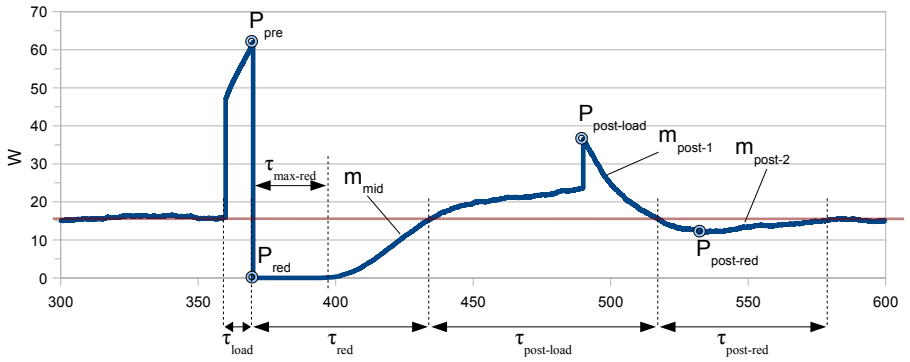


Abbildung 8.3: Kennwerte einer TLR-Lastkurve.

8.1 Signalanalyse: DSC

Das DSC-Signal existiert in zwei Varianten (*load*, *unload*) und bietet darüberhinaus nur den Streuungsparameter spread als Möglichkeit zur Beeinflussung. Es wird daher nun geprüft, inwieweit die resultierende Kurvenform der mittleren Last einer großen Population von Kühlgeräten bei den zwei Signaltypen durch verschiedene Ausprägungen des Parameters spread beeinflusst wird. Für diese Untersuchungen wird das DSC-Signal mit zufallsbasierter Dämpfung in der *full-width*-Variante verwendet (siehe Abbildung 5.4). Die untersuchten Ausprägungen des spread Parameters betrugen spread $= [0, 5, 10, 30, 60, 120]\, min$. Abbildung 8.4 zeigt Simulationen des DSC-*load*-Signals. Das Signal traf in $t_{notify} = 90\, min$ ein. Eine genaue Analyse der sich ergebenden Kurvenverläufe brachte die in Tabelle 8.1 dargestellten Kennwerte. Dabei bezeichnen die Werte $t_{max-load}$ bzw. $p_{max-load}$ den Zeitpunkt sowie die Höhe des Lastmaximums im unmittelbaren Anschluss an t_{notify}. Entsprechend beschreiben die Werte $t_{max-red}$ und $p_{max-red}$ den Zeitpunkt bzw. die Höhe des folgenden Lastminimums, das durch das Steuersignal verursacht wurde. Die Werte $t_{post-load}$ und $p_{post-load}$ wiederum stehen für das zweite Lastmaximum, im Anschluss an das vorangegangene Minimum. Die untere Auflistung zeigt weitere Kennwerte, dabei stehen die mit „m“ bezeichneten Spalten für geschätzte Steigungen (in $\frac{W}{min}$) zwischen den Lastextremen. Die Werte τ_{load}, τ_{red} und $\tau_{post-load}$ beschreiben die Längen der Zeiträume von jeweils erhöhter, verringerter, und anschließend wieder erhöhter Last im Vergleich zum globalen Mittel (vgl. Abbildung 8.1).

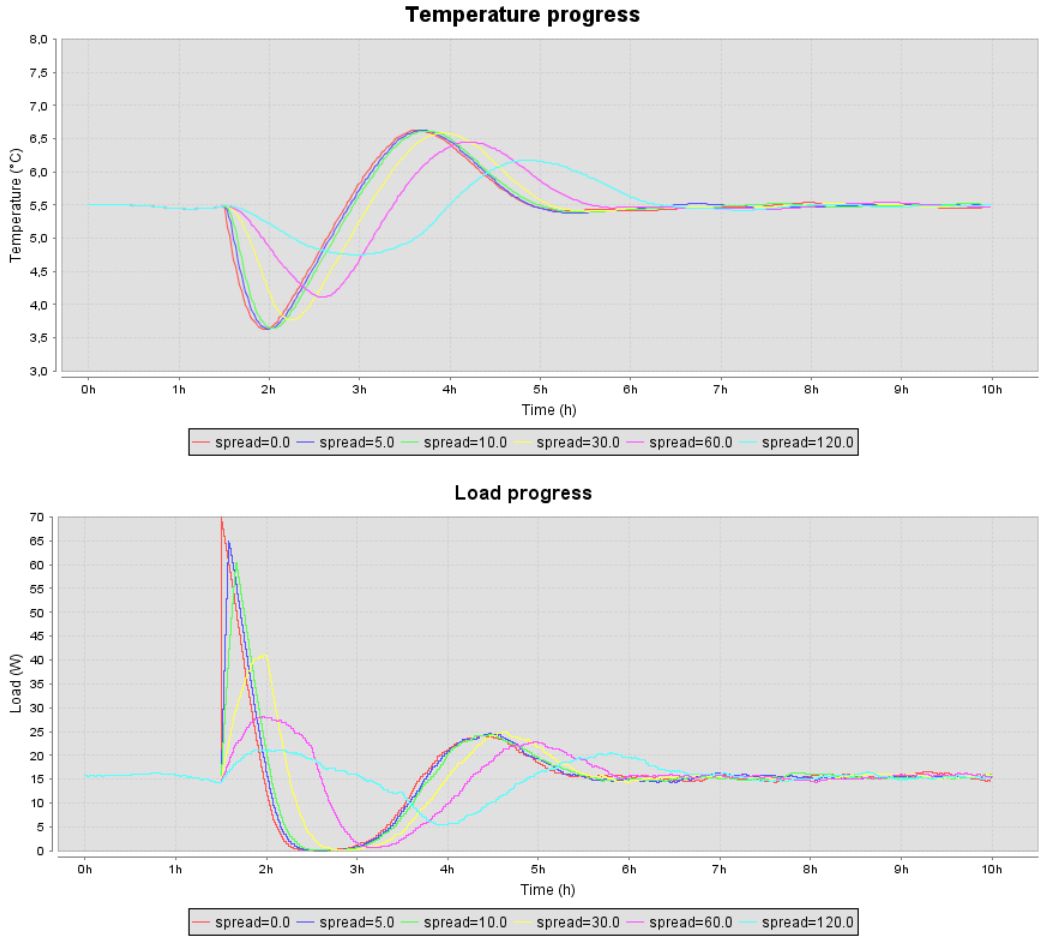


Abbildung 8.4: DSC-*load*-Kurvenverlauf mit verschiedenen spread Werten.

spread	$t_{max-load}$	$p_{max-load}$	$t_{max-red}$	$p_{max-red}$	$t_{post-load}$	$p_{post-load}$
00:00	01:30	70 W	02:33	0 W	04:17	24 W
00:05	01:35	65 W	02:33	0 W	04:26	24.5 W
00:10	01:40	60 W	02:33	0 W	04:26	24.5 W
00:30	02:00	41 W	02:50	0.25 W	04:37	25 W
01:00	01:56	28 W	03:08	0.75 W	04:57	23 W
02:00	01:51	21 W	03:55	5 W	05:48	20.5 W

spread	m_{pre}	m_{mid-1}	m_{mid-2}	m_{post}	τ_{load}	τ_{red}	$\tau_{post-load}$
00:00	∞	-2	0.33	-0.16	00:28	01:40	01:38
00:05	10.12	-2	0.33	-0.18	00:30	01:44	01:45
00:10	4.65	-2	0.33	-0.14	00:33	01:46	01:43
00:30	1.22	-1.73	0.28	-0.16	00:45	01:45	01:33
01:00	0.71	-0.4	0.27	-0.14	01:06	01:39	01:40
02:00	0.33	-0.12	0.18	-0.06	01:31	01:53	02:16

Tabelle 8.1: Kennwerte der DSC-*load*-Simulationen mit verschiedenen spread Werten.

Durch Variation des spread Parameters konnte also eine signifikante Reduktion der Last unter das globale Mittel von minimal $1\,h\,39\,m$ bis maximal $1\,h\,53\,m$ erreicht werden, wobei bei großen spread Werten keine Reduktion auf 0 W erreicht wurde. Die Lastspitze vor der Reduktion schwankt insgesamt im Bereich $[21\,W, 70\,W]$, die Laststeigung nach der Reduktion hingegen nur im Bereich $[20.5\,W, 25\,W]$. Die Steigungen zwischen den Extremen variieren vor der ersten Lastspitze sehr stark im Bereich $[0.33, \infty]\,\frac{W}{min}$, im weiteren Verlauf nimmt die Spannweite der Schwankung stetig ab.
In Abbildung 8.5 sind entsprechend Simulationen des DSC-*unload*-Signales dargestellt, auch hier mit spread $= [0, 5, 10, 30, 60, 120]\,min$. Das Signal wurde wie zuvor in $t_{notify} = 90\,min$ abgesetzt. Die Analyse ergab die in Tabelle 8.2 aufgelisteten Werte (vgl. Abbildung 8.2). Im Gegensatz zu vorheriger Tabelle sind hier weniger Kennwerte enthalten, dies liegt an der simpleren Kurvenform der DSC-*unload*-Simulation. Es existieren lediglich zwei Lastextreme, und demzufolge auch nur drei Steigungen

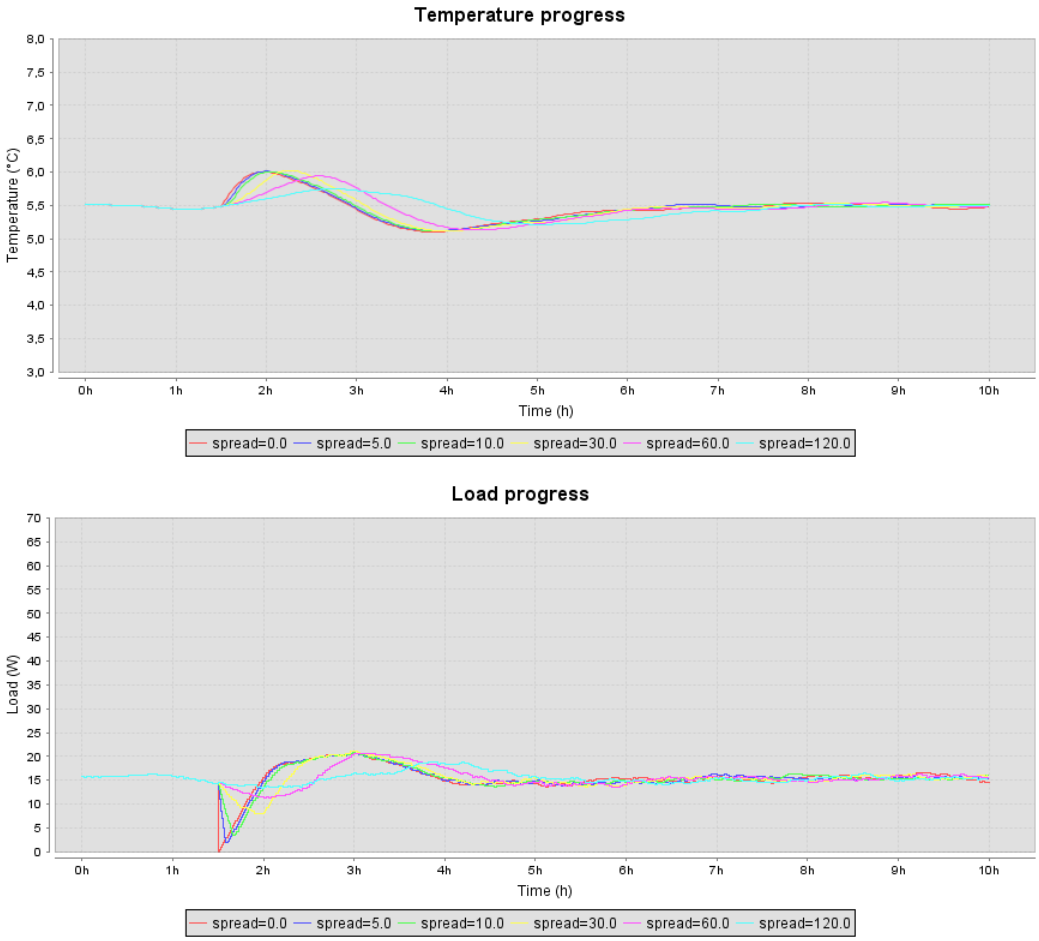


Abbildung 8.5: DSC-*unload*-Kurvenverlauf mit verschiedenen spread Werten.

(vor, zwischen und nach den Extremen) sowie ein Bereich mit verringerter (τ_{red}), und einer mit erhöhter Last (τ_{load}). In der hier durchgeführten Simulation ergaben sich ähnliche Unterschiede in den Kurven wie bei der vorangegangenen. Das erste Lastextrem schwankt recht stark im Bereich $[0\,W, 13.3\,W]$, das zweite wiederum nur im Bereich $[19\,W, 21.25\,W]$. Die Steigungen verhalten sich entsprechend: m_{pre} variiert stark im Bereich $[-0.04, -\infty]\,\frac{W}{min}$, m_{mid} und m_{post} schwanken dafür kaum noch. Insgesamt konnte hier eine Reduktion der Last von minimal $0\,h\,27\,m$ bis maximal $1\,h\,03\,m$ erreicht werden, und die nachfolgende Lasterhöhung bewegte sich zwischen $2\,h\,05\,m$ und $2\,h\,59\,m$.

spread	$t_{max-red}$	$p_{max-red}$	$t_{max-load}$	$p_{max-load}$
00:00	01:30	0 W	02:59	20.75 W
00:05	01:35	1.5 W	02:59	20.75 W
00:10	01:40	3 W	02:59	20.75 W
00:30	01:57	7.75 W	03:00	21.25 W
01:00	02:00	11.25 W	03:11	20.75 W
02:00	01:59	13.3 W	03:51	19 W

spread	m_{pre}	m_{mid}	m_{post}	τ_{red}	τ_{load}
00:00	$-\infty$	0.55	-0.11	00:27	02:05
00:05	-2.85	0.55	-0.09	00:29	02:13
00:10	-1.14	0.55	-0.08	00:31	02:16
00:30	-0.25	0.39	-0.09	00:43	02:00
01:00	-0.11	0.19	-0.07	01:02	02:07
02:00	-0.04	0.05	-0.04	01:03	02:59

Tabelle 8.2: Kennwerte der DSC-*unload*-Simulationen mit verschiedenen spread Werten.

8.2 Signalanalyse: TLR

Das TLR-Signal bietet etwas mehr Möglichkeiten zur Beeinflussung als das DSC-Signal durch die Möglichkeit der Kombination von verschiedenen $\tau_{preload}$ und τ_{reduce} Werten. In Abbildung 8.6 ist zunächst eine Reihe von Simulationsläufen mit $\tau_{preload} = [0, 5, 10, 30, 60, 120]\,min$ zu sehen. Der Zeitpunkt t_{notify} der einzelnen Simulationen wurde so angepasst, dass der Beginn des Reduktionsintervalles immer auf $t_{activ} = 4\,h\,00\,m$ fiel. Die geforderte Länge des Intervalles betrug jeweils $\tau_{reduce} = 2\,h\,00\,m$. Tabelle 8.3 zeigt wiederum die Kennwerte der sich ergebenden Kurven (vgl. Abbildung 8.3). Betrachtet man die Lastkurven bzw. Kennwerte, so fällt als erstes auf, dass der Verlauf für $\tau_{preload} = [60, 120]\,min$ nahezu identisch ist, und die Kurve für $\tau_{preload} = 30\,min$ ebenfalls kaum davon zu unterscheiden ist. Dies deutet darauf hin, dass bei einem geforderten Reduktionsintervall der Länge $\tau_{reduce} = 2\,h\,00\,m$ eine Variation der Vorlaufzeit nur innerhalb des Bereiches $[0, 30]\,min$ sinnvoll ist.

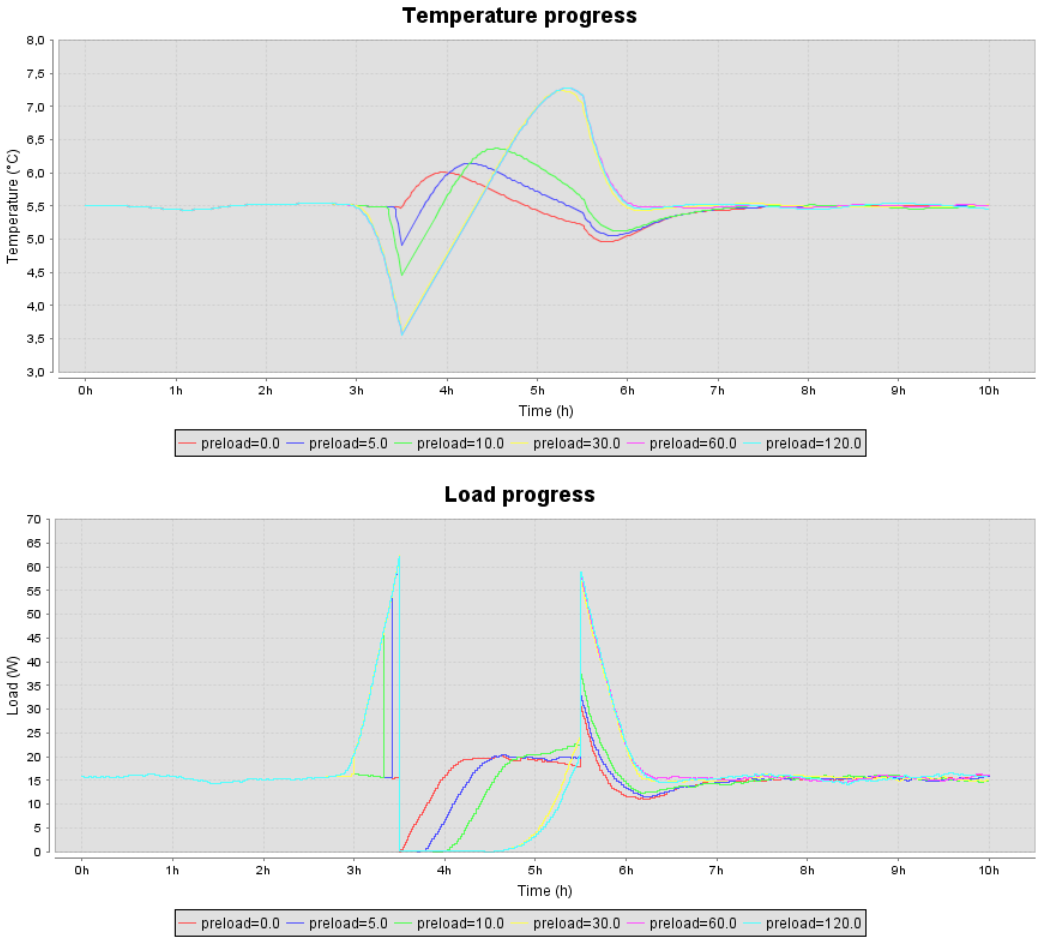


Abbildung 8.6: TLR-Kurvenverlauf mit verschiedenen $\tau_{preload}$ Werten.

Weiterhin ist zu erkennen, dass sich die Breite des Bereiches der erhöhten Last im Vorlaufintervall sowie der mögliche Zeitraum der vollständigen Lastreduktion auf 0 W annähernd linear zur gegebenen Länge $\tau_{preload}$ des Vorlaufintervalles verhalten. Die Höhe dieses Lastmaximums bleibt allerdings bis auf den Fall, dass gar keine Vorlaufzeit gegeben ist, bei jedem verwendeten Parameter konstant bei 62.5 W, nur der Anstieg bis zu diesem Maximum verändert sich je nach $\tau_{preload}$. Die dem Reduktionsintervall folgende Lastspitze ist ebenso umso höher, je länger das Vorlaufintervall war, und auch die darauf folgende Absenkung der Last unter den globalen Mittelwert verhält sich in ihrer Höhe proportional zu $\tau_{preload}$. Die Steigungen der Last in den einzelnen Bereichen zeigen untereinander kaum Schwankungen. Der letzte in der

$\tau_{preload}$	t_{pre}	p_{pre}	t_{red}	p_{red}	$t_{post-load}$	$p_{post-load}$	$t_{post-red}$	$p_{post-red}$
00:00	03:30	0 W	03:30	0 W	05:30	30.5 W	05:06	11 W
00:05	03:30	62.5 W	03:30	0 W	05:30	33 W	06:14	11.5 W
00:10	03:30	62.5 W	03:30	0 W	05:30	37.5 W	06:11	12.25 W
00:30	03:30	62.5 W	03:30	0 W	05:30	57 W	06:24	14.75 W
01:00	03:30	62.5 W	03:30	0 W	05:30	58.5 W	06:23	15.5 W
02:00	03:30	62.5 W	03:30	0 W	05:30	59 W	06:23	14.5 W

$\tau_{preload}$	m_{mid}	m_{post-1}	m_{post-2}	τ_{load}	τ_{red}	$\tau_{post-load}$	$\tau_{post-red}$	$\tau_{max-red}$
00:00	0.18	-0.85	0.08	00:00	00:28	01:48	01:21	00:00
00:05	0.18	-0.85	0.07	00:05	00:46	01:36	01:22	00:12
00:10	0.18	-0.93	0.04	00:10	01:03	01:24	01:40	00:24
00:30	0.07	-1.17	0.01	00:30	01:49	00:51	01:10	00:56
01:00	0.06	-1.21	0.0	00:42	01:53	00:55	00:00	00:56
02:00	0.06	-1.22	0.02	00:42	01:53	00:52	00:40	00:56

Tabelle 8.3: Kennwerte der TLR-Simulationen mit verschiedenen $\tau_{preload}$ Werten.

Tabelle angegebene Kennwert $\tau_{max-red}$ bezeichnet die Dauer, für die die Last auf 0 W abgesenkt werden konnte, und bewegt sich im Bereich $[0, 56]\,min$.

Nun soll anschließend untersucht werden, wie sich die Variation von τ_{reduce} auf den Kurvenverlauf auswirkt. Dazu ist in Abbildung 8.7 eine Reihe von Simulationsläufen mit $\tau_{reduce} = [0, 30, 90, 150]\,min$ dargestellt. Das TLR-Signal wurde in $t_{notify} = 3\,h\,30\,m$ abgesetzt, und das Vorlaufintervall hatte die Länge $\tau_{preload} = 0\,h\,30\,m$. Dadurch begann das Reduktionsintervall wie in der vorherigen Simulationsreihe jeweils in $t_{activ} = 4\,h\,00\,m$. An der Grafik ist zunächst einmal ist zu erkennen, dass die Höhe des Lastmaximums nach dem Reduktionsintervall sich offenbar proportional zur Länge des Intervalles verhält. Dies gilt allerdings nur, solange die Mehrheit der simulierten Geräte das geforderte Reduktionsintervall technisch überhaupt überstehen kann. Die gelbe Lastkurve repräsentiert den Simulationslauf mit $\tau_{reduce} = 2\,h\,30\,m$ und zeigt deutlich, dass die Lastspitze nach der Reduktion bereits

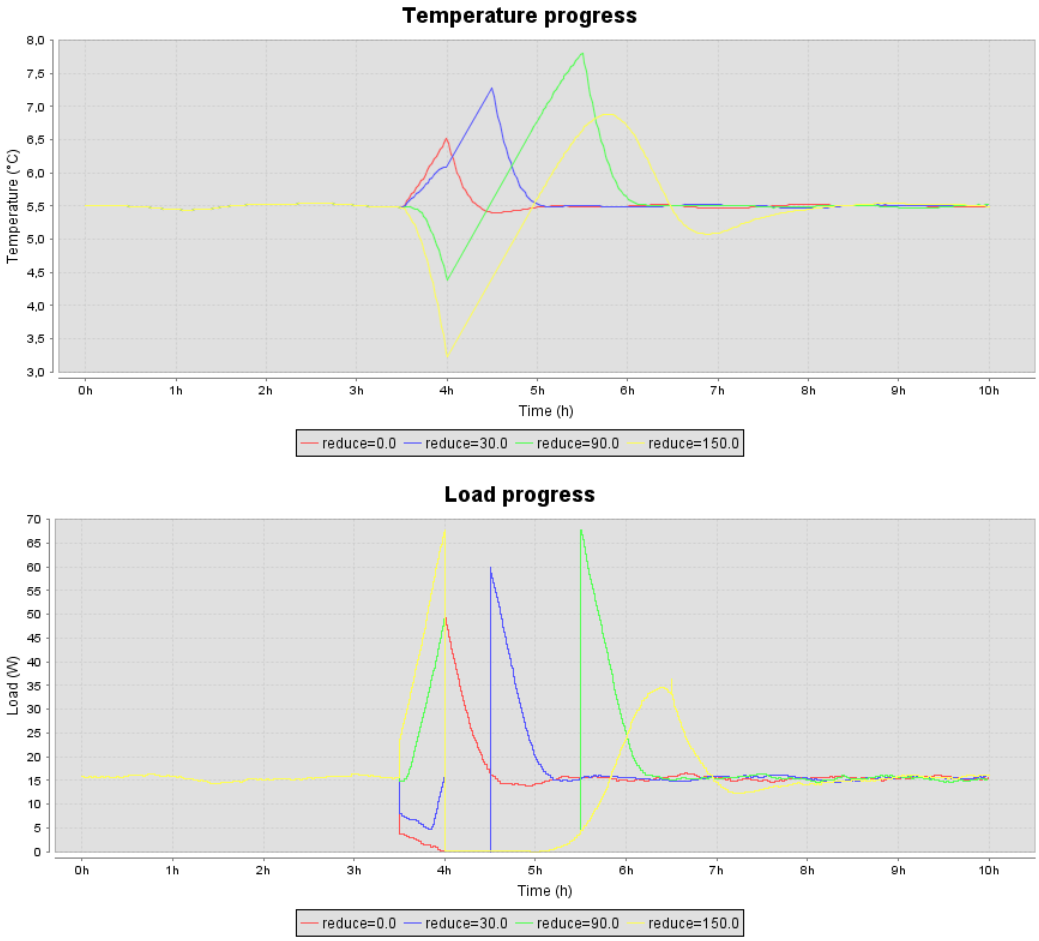


Abbildung 8.7: TLR-Kurvenverlauf mit verschiedenen τ_{reduce} Werten.

in $t_{post-load} = 6\,h\,20\,m$ ihr lokales Maximum hat, das ist 10 Minuten vor Ende des eigentlich geforderten Reduktionsintervalles. Betrachtet man nun das Vorlaufintervall $\tau_{preload}$, so ergibt sich ein auf den ersten Blick merkwürdiges Bild, denn statt die Zeit zum Kühlen zu nutzen, sinken die Lastkurven für die Simulationen mit kleinem τ_{reduce} sogar unter das globalen Mittel ab. Dies liegt an der programmierten Strategie des *optimalen Kühlens*. Diese ist darauf ausgelegt, genau zum Start des Reduktionsintervalles mit einer Temperatur T_{activ} in den Aufwärmbetrieb zu schalten, sodass zum Ende des Intervalles die Maximaltemperatur T_{max} erreicht sein wird. Bei kurzen Reduktionsintervallen bedeutet dies jedoch, dass die optimale Temperatur T_{activ} zum Betreten des Intervalles sehr groß ist, sodass viele Geräte bereits im Vor-

laufintervall mit dem Aufwärmvorgang beginnen müssen, um diese Temperatur zu erreichen. Einerseits lässt sich dieser Effekt nutzen, um eine Lastkurve zu produzieren, die sich wie die hier rote Kurve mit $\tau_{reduce} = 0\,h\,00\,m$ ähnlich einer DSC-*unload* verhält. Andererseits lässt sich aus diesem Verhalten auch ableiten, dass ein Vorlaufintervall mit $\tau_{preload} = 0\,h\,30\,m$ zu lang für kleine Reduktionsintervalle ist, um eine „normale“ TLR-Kurve zu erhalten. Daher ist in Abbildung 8.8 eine weitere Reihe

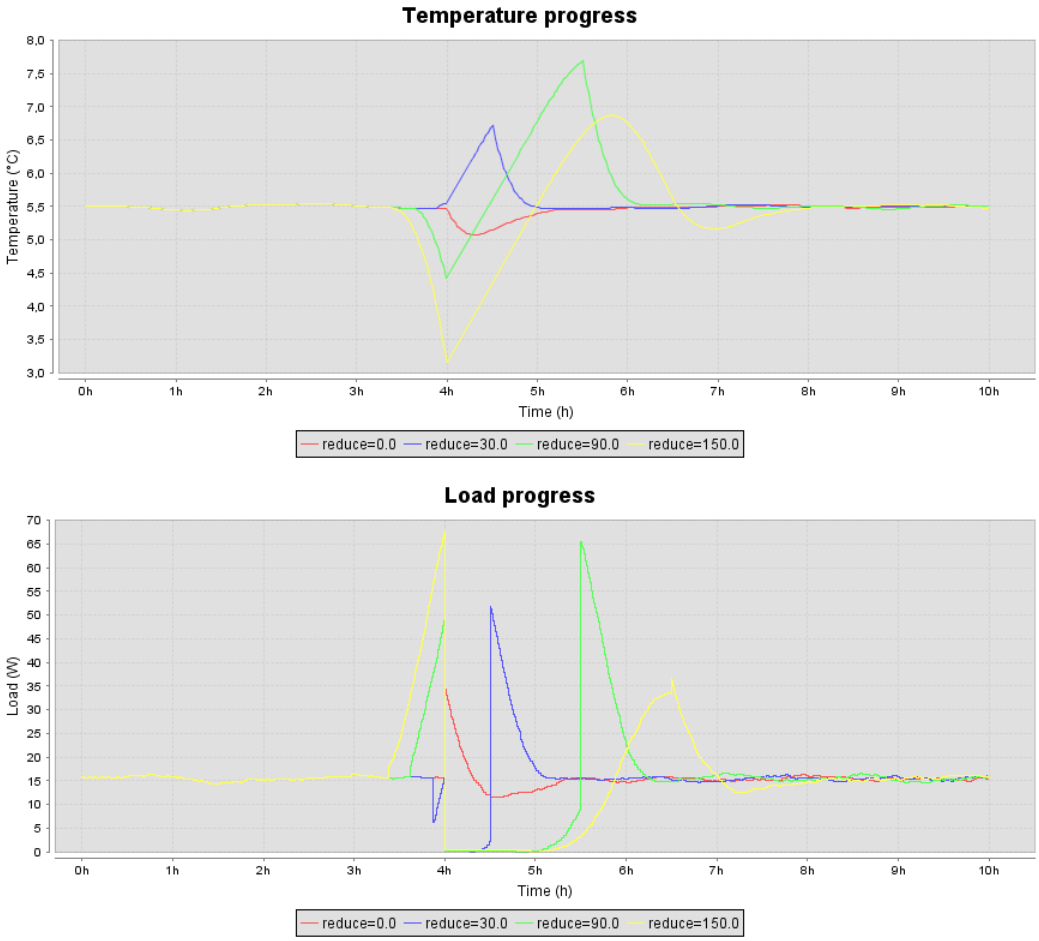


Abbildung 8.8: TLR-Kurvenverlauf mit verschiedenen τ_{reduce} und $\tau_{preload}$ Werten.

von TLR-Simulationsläufen dargestellt, in welchen nicht nur τ_{reduce} variiert wurde, sondern zusätlich auch $\tau_{preload}$. Dabei wurde ausgehend von den bisherigen Simulationen für das vorliegende Szenario ein willkürliches Verhältnis von $\tau_{preload} = \frac{\tau_{reduce}}{4}$ festgelegt. Der Zeitpunkt t_{notify} wurde wiederum so angepasst, dass der Beginn des

Reduktionsintervalles jeweils bei $t_{activ} = 4\,h\,00\,m$ lag. Es ist zu erkennen, dass die einzelnen Kurven denen aus Abbildung 8.7 ähneln, die auffälligsten Unterschiede liegen im nun veränderten Vorlaufintervall der einzelnen Simulationen. Dieses wird nun primär zum Vorkühlen verwendet, wie ursprünglich vorgesehen. Allerdings bedingt diese Anpassung des Wertes $\tau_{preload}$ auch, dass das Reduktionsintervall in den einzelnen Simulationsläufen nicht mehr so exakt eingehalten wird wie vorher. Weiterhin sind die Lastspitzen alle etwas niedriger als bei der Variante mit festem $\tau_{preload}$.
Diese beiden Experimente mit den TLR-Parametern zeigen bereits, dass hier vielfältige Konfigurationsoptionen vorhanden sind. Die einzelnen Möglichkeiten im Einzelnen zu beleuchten würde in diesem Abschnitt der Ideensammlung allerdings zu weit gehen. Daher wird nun im folgenden Abschnitt auf eine weitere, bisher nicht angesprochene Möglichkeit der Erzeugung von Kurvenformen eingegangen.

8.3 Kombination der Steuersignale

Am Anfang dieses Kapitels wurde erwähnt, dass es wünschenswert wäre, durch die Steuersignale einen vordefinierten Kurvenverlauf produzieren zu können. Da die einzelnen Steuerungen trotz ihrer veränderbaren Parameter in sich aber noch recht starr sind, kann dies nicht durch ein einzelnes Signal erreicht werden. Denkbar ist aber eine Kombination der Steuerungen, sodass etwa einzelne Teilgruppen einer großen Population von Kühlgeräten verschiedene Steuersignale erhalten, damit sich im Ganzen eine gewünschte mittlere Lastkurve ergibt. Die Frage ist allerdings, wie die Trennung der Gerätegruppen erfolgen soll. Möglich wäre etwa eine regionale Trennung. So könnte der Netzbetreiber, welcher die Steuersignale absetzt, beispielsweise bestimmten Teilnetzen unterschiedliche Steuersignale schicken, und die adaptiven Geräte in diesen Subnetzen reagieren dann je nach Signal darauf. In der dieser Arbeit zugrunde liegenden Schrift [34] wurde durch die Einführung eines sogenannten *participation ratio* bereits in diese Richtung gedacht. Dieser Parameter sollte definieren, wie viele der potenziellen Signalempfänger ein abgesetztes Steuersignal auch tatsächlich bearbeiten. Die Simulationen zeigten dort, dass ein *participation ratio* < 1 eine lineare Reduktion der entstehenden Lastspitzen und -senken bewirkte. Übertragen auf die Idee der regionalen Abtrennung von Geräten bedeutet dies also, dass neben der Kombination von Signalen auch deren Effektstärke beeinflusst werden kann, indem eben nicht allen Geräten ein Signal gesendet wird. Führt man die Überlegung weiter, so wird klar, dass eine regionale Trennung nicht einmal notwendig ist, um dieses Verhalten zu erzielen. Angenommen, es wird ein Mechanismus in die Controller

eingeführt, welcher auf Basis eines mit dem Signal übertragenen participation Parameters und einer Zufallszahl entscheidet, ob der Controller dem gesendeten Signal Folge leistet. Dann wäre ein Szenario denkbar, in dem die erforderlichen Signaltypen nacheinander mit den ihnen zugeordneten participation Parametern abgesetzt werden, und jeweils nur die dem Parameter entsprechende Menge an Controllern reagiert.
Beispiel: Es sollen 70% der Geräte Signal 1 erhalten, 20% Signal 2 und die restlichen 10% Signal 3, um eine wünschenswerte mittlere Lastkurve zu erzielen. Um dies zu erreichen, könnte zunächst Signal 1 mit $\mathsf{participation}_1 = 0.7$ abgesetzt werden. Alle steuerbaren Geräte erhalten dieses Signal, doch aufgrund des Parameters entscheiden nur 70% der Controller, dem Signal Folge zu leisten, und starten das entsprechende Programm. Anschließend wird Signal 2 abgesetzt. Da jetzt aber schon ein großer Teil der Controller in ihrem jeweiligen Steuerprogramm sind und nicht mehr auf weitere Signale reagieren, müssen die verbleibenden Prozentzahlen angepasst werden. Die noch zur Verfügung stehende Population soll im Verhältnis $20 : 10$ aufgeteilt werden, das entspricht prozentual $66\% : 33\%$. Somit wird für das zweite Signal der Parameter $\mathsf{participation}_2 = 0.66$ gesetzt. Alle nun noch übrigen Geräte sollen Signal 3 erhalten, daher ergibt sich $\mathsf{participation}_3 = 1.0$.
Auf diese Weise könnten also Steuersignale kombiniert werden, um durch Überlagerung der im Einzelnen möglichen Lastkurven eine spezifische Kurvenform zu erhalten. Es bleibt allerdings zu untersuchen, inwieweit mit den vorliegenden Signaltypen beliebige Verläufe generiert werden können. Konkret bedeutet dies, dass eine Analyse durchgeführt werden müsste, die die bisher produzierbaren Lastkurven daraufhin untersucht, ob sie im mathematischen Sinn ein Erzeugendensystem (siehe etwa [25, S. 191]) bilden. Wenn das der Fall wäre und eine möglichst minimale Menge von Basiskurven als Erzeugendensystem gefunden wurde, könnten durch wiederholte Skalierung und Überlagerung der Basiselemente beliebige Verläufe produziert werden. Dadurch wäre es dann möglich, die durch variierende Erzeugung sowie wechselnden Verbrauch hervorgerufenden Schwankungen im Stromnetz ohne die Hilfe von Augleichskraftwerken auszugleichen, solange diese Schwankungen das Lastverschiebungspotenzial der Kühlgeräte nicht übersteigt. Zumindest würde dieses Potenzial aber eine signifikante Unterstützung im Ausgleichsprozess darstellen.

Kapitel 9

Fazit

Bereits in dem dieser Arbeit zugrundeliegenden Artikel [34] wurde deutlich, dass die Möglichkeit der gesteuerten Lastverschiebung durch adaptive Kühlgeräte in Privathaushalten ein durchaus bemerkenswertes Potenzial zur Lastregulation bietet. Die in der vorliegenden Arbeit entwickelten Mechanismen zur Dämpfung (Kapitel 5) der aus den vorgeschlagenen Steuerungen entstehenden Synchronisation der Gerätezustände liefern darüberhinaus einen Beitrag zur Verwirklichung dieses noch theoretischen Konstrukts. Weiterhin wurden die durchgeführten Simulationen unter der Voraussetzung von inhomogenen Gerätepopulationen durchgeführt, um die Möglichkeiten und Auswirkungen der Steuerungen in einem realen Umfeld vorauszusagen (Kapitel 6). Abschließend bietet die vorgenomme Umsetzung der Simulationsmodelle in endliche Automaten (Kapitel 7) eine grundlegende Basis zur Durchführung von Feldversuchen, in denen die auf theoretischer Ebene gewonnenen Erkenntnisse als Fortsetzung dieser Arbeit in einer praktischen Umsetzung verifiziert werden können.

Es wurden zum Erreichen der soeben beschriebenen Ziele im Laufe der Arbeit zwei voneinander unabhängige Softwareprojekte angelegt. Das erste verwendet das anfangs ausgewählte Simulationsframework SimKit zur Umsetzung der ereignisbasierten Modelle aus dem Hauptteil dieser Arbeit in ausführbare Simulationen inklusive textueller sowie grafischer Auswertung. Dieses Projekt wurde zudem mit einer grafischen Benutzungsoberfläche ausgestattet und über die *Java Web Start* Technologie im Laufe des Entwicklungsprozesses online ausführbar gemacht, um den Gutachtern sowie interessierten Lesern die Bedienung zu erleichtern. Dieses Projekt besteht inklusive Simulationsmodellen, grafischer Auswertung und GUI insgesamt aus etwa 9000 Zeilen Code in 53 Klassen.

Das zweite Projekt enthält das im Schlussteil der Arbeit entwickelte Framework zur Simulation von endlichen Automaten sowie die darin erstellten verschiedenen Controllertypen. Diese Software besteht inklusive Framework, Controller, virtuellem Kühlgerät und grafischer Auswertung insgesamt aus knapp 6000 Zeilen Code in 118 Klassen. Die große Anzahl der Klassen ergibt sich dadurch, dass für alle Zustände und Transitionen jedes Controllers eigene Klassen angelegt wurden, um die einzelnen Controller möglichst klar voneinander zu trennen und Interaktionsfehler auszuschließen. Da diese Software ursprünglich lediglich zur Validierung der Automaten implementiert wurde enthält sie bisher keine grafische Benutzungsoberfläche. Beide Projekte sind in *Java* geschrieben und strukturell in Pakete unterteilt. Es wurden die üblichen Konventionen zur Benennung der Klassen, Methoden und Variablen sowie zur Dokumentierung des Codes eingehalten. In den zuoberst liegenden Paketen sind in jedem Projekt mehrere Einsteigspunkte (Klassen mit *main*-Methoden) der Software enthalten. Diese beinhalten die einzelnen Möglichkeiten zur Erzeugung der in dieser Arbeit vorgestellten Simulationen.

Neben den erfolgreich erreichten Zielen aus dem ersten Absatz dieses Kapitels gibt es in der Arbeit aber auch Punkte, die eine weitere Untersuchung erfordern. So ist in Abschnitt 7.4 beispielsweise deutlich geworden, dass die ursprünglich als nahezu Seiteneffekt-frei angenommene lineare Annäherung der Modellgleichungen aus Abschnitt 3.3 in der Realisierung in endlichen Automaten leider wesentlich stärkere Auswirkungen hat als erwartet. Es wurde gezeigt, dass die vorgestellten Mechanismen trotz der daraus resultierenden Berechnungsfehler korrekt arbeiten, jedoch wäre hier eine Verfeinerung der Modelle zur Minimierung dieser Fehler wünschenswert. Die durch diese Fehler verursachten verkürzten Aufwärmphasen der Controller und die daraus resultierenden Abweichungen ließen sich beispielsweise vermutlich durch eine Anpassung des Basis-Controllers beheben. Dieser könnte etwa, sobald die übergebene Zieltemperatur einer der Grenztemperaturen T_{min} oder T_{max} entspricht, die Berechnung eines τ_{switch}-Wertes unterlassen und den nächsten Phasenwechsel allein dem iterierenden Zustand c_polling bzw. w_polling überlassen (vlg. Abbildung 7.2). Dies könnte die schwerwiegendsten Auswirkungen der Linearisierung ausgleichen.

Eine weitere denkbare Fortsetzung dieser Arbeit wurde im vorangegangenen Kapitel 8 deutlich. Dort wurde auf die Möglichkeit der Kombination verschiedener Steuersignale zur Erzeugung von beliebig gestalteten Lastkurven eingegangen. Im Zusammhang mit der Suche nach einem minimalen Erzeugendensystem von Steuer-

signalen soll hier als Anregung auf das Thema *Wavelets* verwiesen werden. Dieses Themenfeld beschäftigt sich unter anderem mit der Möglichkeit, beliebige Kurvenformen durch eine Menge von Basiskurven zu generieren. Die Anwendungsmöglichkeiten liegen dabei hauptsächlich in der Signalverarbeitung sowie Datenkompression, jedoch könnte dieser Ansatz auch für die vorliegende Fragestellung bedeutsam sein. Eine Einführung in *Wavelets* inklusive Enstehung, Theorie sowie Anwendbarkeit ist beispielsweise in [16] gegeben.

Danksagung

Abschließend möchte ich allen Menschen danken, die diese Arbeit und damit den Abschluss meines Studiums erst möglich gemacht haben. Hervorzuheben ist hier meine geliebte Freundin Daniela, die mich während der letzten sechs Monate permanent umsorgt und mich auch in den Phasen der Ratlosigkeit meisterhaft unterstützt hat. Ihr verdanke ich, dass ich die Ausgeglichenheit bewahren konnte, die in dieser Zeit notwendig war.
Weiterhin möchte ich all meinen Freunden danken, die sich von der Begeisterung für dieses Thema anstecken ließen und eine Woche lang akribische Aufzeichnungen über die Nutzung ihrer Kühlschränke angefertigt haben. Allen voran hier meine bemerkenswerte Schwester Corinna, die jeden Tag aufs Neue über sich hinaus wächst und auch nichts anderes verdient hat. Vielen Dank aber auch an Martin, Martina, Andreas, Anja, Kathi, Christian N., Mark, Christian E., Michael, Jessica, sowie Sarah, Anna und Caro.
Danken möchte ich auch meinen Eltern sowie Josef und Elke. Ohne eure Unterstützung jedweder Art wäre dies hier nicht entstanden.

Dank gebührt natürlich auch meinen Betreuern M. Sonnenschein und U. Vogel, ohne die diese Arbeit letztlich gar nicht erst denkbar gewesen wäre. Ihr freundliches Interesse, die fachkundige Unterstützung und die vielen anregenden Gespräche haben außerdordentlich zur Gestaltung dieser Arbeit beigetragen und sind somit ein Teil von ihr geworden. Ich freue mich auf eine weitere Zusammenarbeit auf dieser Basis.

After all, there is cake...

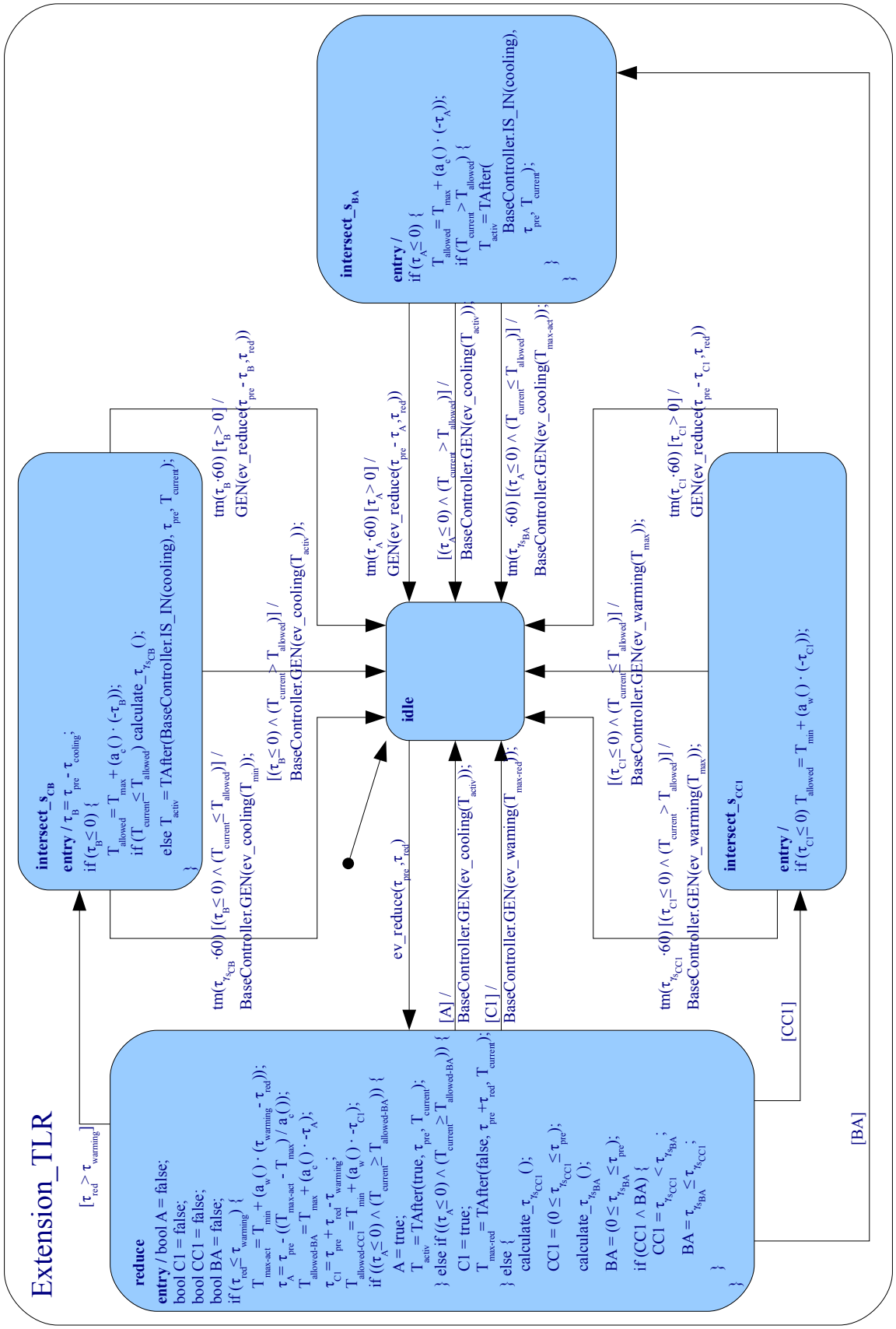
Anhang

Anhang A

Zustandsautomat TLR

Die folgende Grafik zeigt die vollständige Version des Zustandsautomaten für die TLR-Steuerung.
Die verwendeten Funktionen calculate_tau_gamma_scc1(), calculate_tau_gamma_sba() und calculate_tau_gamma_scb() ermitteln die zeitlichen Abstände zu den Schnittpunkten des aktuellen Temperaturverlaufes mit den aus Abschnitt 4.2 bekannten Geraden der Bereichskarte der TLR-Klasseneinteilung (siehe dazu Abbildung 4.6 und Abbildung 4.7). Der reale Temperaturverlauf des Gerätes wird dabei linear angenähert, um die Berechnungen zu vereinfachen. Die Algorithmen dieser Berechnungen sind im Anschluss an die folgende Grafik als Code-Listings dargestellt. Die in diesen Fragmenten verwendete Methode Geometry.findLineSegmentIntersection(...) entstammt der unter der LGPL lizensierten Klasse Geometry von J. Dreyer[1] und ermittelt den Schnittpunkt zweier Geradensegmente.

[1] siehe http://geosoft.no/software/index.html, zuletzt besucht: 24. Oktober 2008


Extension_TLR
reduce
entry / bool A = false;
bool C1 = false;
bool CC1 = false;
bool BA = false;
if (τ_red ≤ τ_warming) {
T_max-act = T_min + (a_w() · (τ_warming - τ_red));
τ_A = τ_pre - ((T_max-act - T_max) / a_c());
T_allowed-BA = T_max + (a_c() · -τ_A);
τ_C1 = τ_pre + τ_red - τ_warming;
T_allowed-CC1 = T_min + (a_w() · -τ_C1);
if ((τ_A ≤ 0) ∧ (T_current ≥ T_allowed-BA)) {
A = true;
T_activ = TAfter(true, τ_pre, T_current);
} else if ((τ_A ≤ 0) ∧ (T_current ≥ T_allowed-BA)) {
C1 = true;
T_max-red = TAfter(false, τ_pre +τ_red, T_current);
} else {
calculate_τ_sCC1 ();
CC1 = (0 ≤ τ_sCC1 ≤ τ_pre);
calculate_τ_sBA ();
BA = (0 ≤ τ_sBA ≤ τ_pre);
if (CC1 ∧ BA) {
CC1 = τ_pre < τ_sCC1 ;
BA = τ_sBA ≤ τ_sCC1 ;
}
}
[τ_red > τ_warming]
intersect_s_CB
entry / τ_B = τ_pre - τ_cooling;
if (τ_B ≤ 0) {
T_allowed = T_max + (a_c() · (-τ_B));
if (T_current ≤ T_allowed) calculate_τ_sCB ();
else T_activ = TAfter(BaseController.IS_IN(cooling), τ_pre, T_current);
}
tm(τ_sCB ·60) [(τ_B ≤ 0) ∧ (T_current ≤ T_allowed)] /
BaseController.GEN(ev_cooling(T_min));
tm(τ_B ·60) [τ_B > 0] /
GEN(ev_reduce(τ_pre - τ_B, τ_red))
[(τ_B ≤ 0) ∧ (T_current > T_allowed)] /
BaseController.GEN(ev_cooling(T_activ));
intersect_s_BA
entry /
if (τ_A ≤ 0) {
T_allowed = T_max + (a_c() · (-τ_A));
if (T_current > T_allowed) {
T_activ = TAfter(
BaseController.IS_IN(cooling),
τ_pre, T_current);
}
}
tm(τ_A ·60) [τ_A > 0] /
GEN(ev_reduce(τ_pre - τ_A, τ_red))
[(τ_A ≤ 0) ∧ (T_current > T_allowed)] /
BaseController.GEN(ev_cooling(T_activ));
tm(τ_sBA ·60) [(τ_A ≤ 0) ∧ (T_current ≤ T_allowed)] /
BaseController.GEN(ev_cooling(T_max-act));
idle
ev_reduce(τ_pre, τ_red)
[A] /
BaseController.GEN(ev_cooling(T_activ));
[C1] /
BaseController.GEN(ev_warming(T_max-red));
[(τ_C1 ≤ 0) ∧ (T_current ≤ T_allowed)] /
BaseController.GEN(ev_warming(T_max));
tm(τ_sCC1 ·60) [(τ_C1 ≤ 0) ∧ (T_current > T_allowed)] /
BaseController.GEN(ev_warming(T_max));
tm(τ_C1 ·60) [τ_C1 > 0] /
GEN(ev_reduce(τ_pre - τ_C1, τ_red))
intersect_s_CC1
entry /
if (τ_C1 ≤ 0) T_allowed = T_min + (a_w() · (-τ_C1));
[CC1]
[BA]

Das folgende Listing beschreibt die Funktion calculate_tau_gamma_scc1(), die den Schnittpunkt der aktuellen Temperaturkurve eines Gerätes mit der Trenngeraden s_{CC1} berechnet.

Listing A.1: Funktion calculate_tau_gamma_scc1($\tau_{preload}, \tau_{reduce}$)

```
1  public double calculate_tau_gamma_scc1() {
2    double now = (double) bc.getClock() / 60.0;
3    double tAct = now + tau_preload;
4    double tauW = bc.getTauWarming();
5    double tauC = bc.getTauCooling();
6    double tC1 = tAct + tau_reduce - tauW;
7    double aw = bc.aw();
8    double TMin = bc.getTmin();
9    double TMax = bc.getTmax();
10   double T_maxact = TMin + (aw * (tAct - tC1));
11   double TCurrent = bc.getFridge().getTemperature();
12   // Find intersection with sCC1
13   Line s = new Line();
14   s.x1 = tC1;
15   s.y1 = TMin;
16   s.x2 = tAct;
17   s.y2 = T_maxact;
18   Line c = new Line();
19   if (!bc.getFridge().isActive()) {
20     c.x1 = now + tau(TCurrent, TMax);
21     c.y1 = TMax;
22     c.x2 = c.x1 + tauC;
23     c.y2 = TMin;
24   } else {
25     c.x1 = now;
26     c.y1 = TCurrent;
27     c.x2 = c.x1 + tau(c.y1, TMin);
28     c.y2 = TMin;
29   }
30   double[] is = new double[2];
31   int code = Geometry.findLineSegmentIntersection(c, s, is);
32   if (code == 1 || code == 0) {
33     return is[0] - now;
34   } else {
35     // No intersection found.
36     return Double.NaN;
37   }
38 }
```

Dieses Listing beschreibt die Funktion calculate_tau_gamma_sba(), die den Schnittpunkt der aktuellen Temperaturkurve eines Gerätes mit der Trenngeraden s_{BA} berechnet.

Listing A.2: Funktion calculate_tau_gamma_sba($\tau_{preload}, \tau_{reduce}$)

```
1  public double calculate_tau_gamma_sba() {
2    double now = (double) bc.getClock() / 60.0;
3    double tAct = now + tau_preload;
4    double tauW = bc.getTauWarming();
5    double tC1 = tAct + tau_reduce - tauW;
6    double aw = bc.aw();
7    double ac = bc.ac();
8    double TMin = bc.getTmin();
9    double TMax = bc.getTmax();
10   double T_maxact = TMin + (aw * (tAct - tC1));
11   double TCurrent = bc.getFridge().getTemperature();
12   double tA = tAct - ((T_maxact - TMax) / ac);
13   // Find intersection with sBA
14   Line s = new Line();
15   s.x1 = tA;
16   s.y1 = TMax;
17   s.x2 = tAct;
18   s.y2 = T_maxact;
19   Line c = new Line();
20   if (bc.getFridge().isActive()) {
21     c.x1 = now + tau(TCurrent, TMin);
22     c.y1 = TMin;
23     c.x2 = c.x1 + tauW;
24     c.y2 = TMax;
25   } else {
26     c.x1 = now;
27     c.y1 = TCurrent;
28     c.x2 = tAct;
29     c.y2 = TAfter(false, tau_preload, TCurrent);
30   }
31   double[] is = new double[2];
32   int code = Geometry.findLineSegmentIntersection(c, s, is);
33   if (code == 1 || code == 0) {
34     return is[0] - now;
35   } else {
36     // No intersection found.
37     return Double.NaN;
38   }
39 }
```

Dieses Listing beschreibt abschließend die Funktion calculate_tau_gamma_scb(), die den Schnittpunkt der aktuellen Temperaturkurve eines Gerätes mit der Trenngeraden s_{CB} berechnet.

Listing A.3: Funktion calculate_tau_gamma_scb($\tau_{preload}, \tau_{reduce}$)

```
1  public double calculate_tau_gamma_scb() {
2    double now = (double) bc.getClock() / 60.0;
3    double tAct = now + tau_preload;
4    double tauW = bc.getTauWarming();
5    double tauC = bc.getTauCooling();
6    double tB = tAct - tauC;
7    double TMin = bc.getTmin();
8    double TMax = bc.getTmax();
9    double TCurrent = bc.getFridge().getTemperature();
10   // Find intersection with sCB
11   Line s = new Line();
12   s.x1 = tB;
13   s.y1 = TMax;
14   s.x2 = tAct;
15   s.y2 = TMin;
16   Line c = new Line();
17   if (bc.getFridge().isActive()) {
18     c.x1 = now + tau(TCurrent, TMin);
19     c.y1 = TMin;
20     c.x2 = c.x1 + tauW;
21     c.y2 = TMax;
22   } else {
23     c.x1 = now;
24     c.y1 = TCurrent;
25     c.x2 = c.x1 + tau(c.y1, TMax);
26     c.y2 = TMax;
27   }
28   double[] is = new double[2];
29   int code = Geometry.findLineSegmentIntersection(c, s, is);
30   if (code == 1 || code == 0) {
31     return is[0] - now;
32   } else {
33     // No intersection found.
34     return Double.NaN;
35   }
36 }
```

Anhang B

Algorithmen der Zustandswiederherstellung

Da die Algorithmen der Zustandswiederherstellung für die Automatenmodelle aus Kapitel 7 unter anderem sehr viele Fallunterscheidungen enthalten, wurden sie nicht direkt in die entsprechenden Diagramme der vorliegenden Arbeit integriert. Stattdessen werden sie hier als Code-Listings abgebildet, welche dem Java-Code der entwickelten Simulationssoftware für Zustandsautomaten auf beiliegender CD-ROM entnommen wurden. Auch hier wird die bereits in Anhang A vorgestellte Klasse Geometry von J. Dreyer verwendet.
Das folgende Listing beschreibt die Methode currentState(), die ein neues Objekt DeviceState erzeugt, welches den aktuellen Zustand des Kühlgerätes widerspiegelt.

Listing B.1: Methode currentState()

```
1 public void currentState() {
2   return new DeviceState(ext.getBc().getFridge().getTemperature(),
3         ext.getBc().getFridge().isActive());
4 }
```

Das nächste Listing repräsentiert die Funktion stateAfter(DeviceState current, double timespan) aus Abbildung 7.7 in seiner Entsprechung im Java Code der Simulationssoftware. Sie berechnet den fiktiven zukünftigen Zustand des betreffenden Gerätes,

den es ausgehend vom übergebenen Startzustand current nach der ebenfalls übergebenen Zeitspanne timespan hätte.

Listing B.2: Methode stateAfter(DeviceState current, double timespan)

```
/**
 * Calculates the state of the fridge how it would be after the given
 * timespan from now on. The timespan may include phase changes. The
 * parameter <code>current</code> determines the state of the device at the
 * current point of time.
 *
 * @param current
 * @param timespan
 * @return
 */
public DeviceState stateAfter(DeviceState current, double timespan) {
  // Calculate base timespans
  double tau_phaseChange = tlr.tau(current.t, current.active ? bc
      .getTmin() : bc.getTmax());
  DeviceState ret = new DeviceState();
  if (timespan <= tau_phaseChange) {
    // No phase change occurs in the given timespan.
    ret.active = current.active;
    ret.t = tlr.TAfter(current.active, timespan, current.t);
  } else {
    // At least one phase change will occur in given timespan.
    // ts = remaining timespan after first phase change
    double ts = timespan - tau_phaseChange;
    // c = amount of complete full cycles (tw+tc) in ts
    int c = (int) Math.floor(ts
        / (bc.getTauWarming() + bc.getTauCooling()));
    // remainder = remaining time in ts after last complete full cycle
    double remainder = ts
        - ((bc.getTauWarming() + bc.getTauCooling()) * (double) c);
    // Check type of the phase after timespan and determine resulting
    // state
    if (current.active) {
      // We started with cooling, so the first *full* phase began with
      // warming.
      if (remainder < bc.getTauWarming()) {
        // The remainder cannot contain a whole warming phase, so we
        // will still be warming after the timespan.
        ret.active = false;
        ret.t = tlr.TAfter(false, remainder, bc.getTmin());
      } else {
        // The remainder contains a whole warming phase, so we will
        // be cooling after the timespan.
        ret.active = true;
        ret.t = tlr.TAfter(true, remainder - bc.getTauWarming(), bc
            .getTmax());
      }
```

```
47      } else {
48        // We started with warming, so the first *full* phase began with
49        // cooling.
50        if (remainder < bc.getTauCooling()) {
51          // The remainder cannot contain a whole cooling phase, so we
52          // will still be cooling after the timespan.
53          ret.active = true;
54          ret.t = tlr.TAfter(true, remainder, bc.getTmax());
55        } else {
56          // The remainder contains a whole cooling phase, so we will
57          // be warming after the timespan.
58          ret.active = false;
59          ret.t = tlr.TAfter(false, remainder - bc.getTauCooling(),
60              bc.getTmin());
61        }
62      }
63    }
64    return ret;
65  }
```

Das folgende Listing beschreibt abschließend die Funktion der in Abbildung 7.7 verwendeten Anweisung γ(rState, currentState()).

Listing B.3: Methode gamma(DeviceState s1, DeviceState s2)

```
1   /**
2    * Calculates the delay to the first point in future time where the
3    * temperature curves of the two given DeviceStates will cross.
4    *
5    * @param s1
6    * @param s2
7    * @return
8    */
9   public double gamma(DeviceState s1, DeviceState s2) {
10    double[] is = new double[2];
11    int ret;
12    // Create line segments
13    Line mod = new Line();
14    Line reg = new Line();
15    Line modNext = new Line();
16    Line regNext = new Line();
17    mod.x1 = 0;
18    mod.y1 = s1.t;
19    if (s1.active) {
20      mod.x2 = mod.x1
21          + tlr.tau(s1.t, bc.getTmin());
22      mod.y2 = bc.getTmin();
23      modNext.x1 = mod.x2;
24      modNext.y1 = mod.y2;
```

```
25      modNext.x2 = modNext.x1
26          + tlr.tau(modNext.y1, bc.getTmax());
27      modNext.y2 = bc.getTmax();
28    } else {
29      mod.x2 = mod.x1
30          + tlr.tau(s1.t, bc.getTmax());
31      mod.y2 = bc.getTmax();
32      modNext.x1 = mod.x2;
33      modNext.y1 = mod.y2;
34      modNext.x2 = modNext.x1
35          + tlr.tau(modNext.y1, bc.getTmin());
36      modNext.y2 = bc.getTmin();
37    }
38    reg.x1 = 0;
39    reg.y1 = s2.t;
40    if (s2.active) {
41      reg.x2 = reg.x1
42          + tlr.tau(s2.t, bc.getTmin());
43      reg.y2 = bc.getTmin();
44      regNext.x1 = reg.x2;
45      regNext.y1 = reg.y2;
46      regNext.x2 = regNext.x1
47          + tlr.tau(regNext.y1, bc.getTmax());
48      regNext.y2 = bc.getTmax();
49    } else {
50      reg.x2 = reg.x1
51          + tlr.tau(s2.t, bc.getTmax());
52      reg.y2 = bc.getTmax();
53      regNext.x1 = reg.x2;
54      regNext.y1 = reg.y2;
55      regNext.x2 = regNext.x1
56          + tlr.tau(regNext.y1, bc.getTmin());
57      regNext.y2 = bc.getTmin();
58    }
59    // Check intersection of mod & reg
60    ret = Geometry.findLineSegmentIntersection(mod, reg, is);
61    if (ret != 1) {
62      // Check intersection of mod & regNext
63      ret = Geometry.findLineSegmentIntersection(mod, regNext, is);
64    }
65    if (ret != 1) {
66      // Check intersection of modNext & reg
67      ret = Geometry.findLineSegmentIntersection(modNext, reg, is);
68    }
69    if (ret != 1) {
70      // Check intersection of modNext & regNext
71      ret = Geometry.findLineSegmentIntersection(modNext, regNext, is);
72    }
73    if (ret != 1) {
74      // Should not happen:
75      System.err.println(getName() + " - No crossing point found: " + s1
```

```
76          + " vs. " + s2);
77    }
78    // Check rounding error (negative delay)
79    if (is[0] < 0) {
80      System.err
81          .println("Negative delay fixed from " + is[0] + " to 0.0");
82      return 0.0;
83    } else {
84      return is[0];
85    }
86  }
```

Anhang C

Zusätzliche FSM-Simulationsergebnisse

Da ein Controller für das Steuersignal TLR mit der in dieser Arbeit verfolgten Strategie des *optimalen Kühlens* (vgl. Abschnitt 4.2.1) sehr viele Fallunterscheidungen berücksichtigen muss und sich je nach Gegebenheiten anders verhält, ergeben sich je nach verwendeten Parametern eine Menge an verschiedenen Simulationsergebnissen. Damit diese den Lesefluss nicht stören, wurden speziell in Abschnitt 7.4 nur exemplarische Ergebnisse dargestellt. Weitere Zeitreihen sind daher hier zusammengefasst, die folgenden Grafiken beziehen sich auf die ungedämpfte TLR-Steuerung aus Abbildung 7.6 bzw. Anhang A.

Abbildung C.1 zeigt die Ergebnisse einer Simulation mit $t_{notify} = 2\,h\,30\,min$, $\tau_{preload} = 30\,min$ und $\tau_{reduce} = 1\,h\,30\,min$. Das Gerät wurde hier in die Klasse C eingeordnet, und befand sich zu diesem Zeitpunkt in der Kühlphase. Daher trat ein Schnittpunkt mit der Geraden s_{CC1} auf und der Automat wählte den Weg über den Zustand intersect_s_{CC1}.

In Abbildung C.2 wurde das Gerät in die Klasse A eingeordnet, da es zum Zeitpunkt $t_{notify} = 2\,h$ eine zu hohe Temperatur hatte, um innerhalb des Intervalles $\tau_{preload} = 10\,min$ genügend herunterzukühlen. Es kühlt daher solange es kann bis zum Beginn des Reduktionsintervalls, um dieses möglichst lange zu überstehen. Diese Situation stellt den ersten Sonderfall aus Abbildung 7.6 dar, in welchem direkt vom Zustand reduce aus wieder zurück in den Zustand idle gewechselt wird.

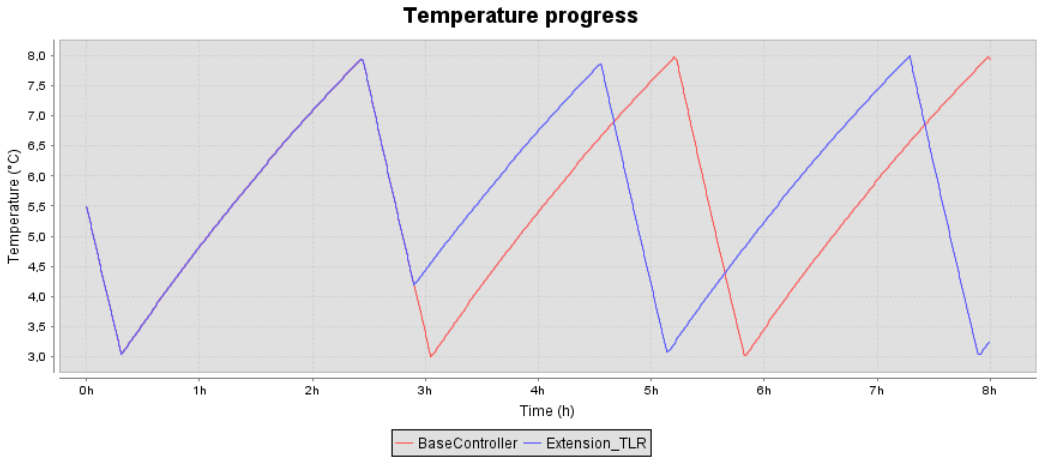


Abbildung C.1: Temperaturverlaufssimulationen der FSM BaseController und Extension_TLR mit $t_{notify} = 2\,h\,30\,min,\ \tau_{preload} = 30\,min,\ \tau_{reduce} = 1\,h\,30\,min.$

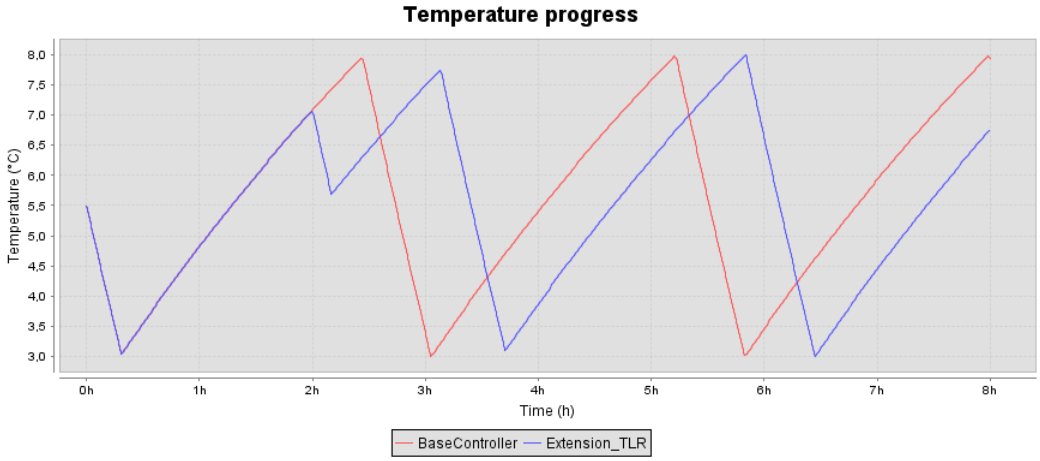


Abbildung C.2: Temperaturverlaufssimulationen der FSM BaseController und Extension_TLR mit $t_{notify} = 2\,h,\ \tau_{preload} = 10\,min,\ \tau_{reduce} = 1\,h\,30\,min.$

Der zweite Sonderfall dieser Art ist in Abbildung C.3 dargestellt. Hier ist das Gerät schon im Zeitpunkt $t_{notify} = 30\,min$ genügend gekühlt, um das Reduktionsintervall vollständig zu überstehen. Es wurde daher in die Klasse C1 eingeordnet. Der Automat hat im Zustand reduce die Zieltemperatur $T_{max-red}$ berechnet, welche zum Ende des Reduktionsintervalles erreicht sein wird und diese als Zieltemperatur an den Basis-Controller gesendet. Dadurch fand unmittelbar nach der Reduktion ein Phasenwechsel statt, genau wie es die Strategie des optimalen Kühlens vorschreibt.

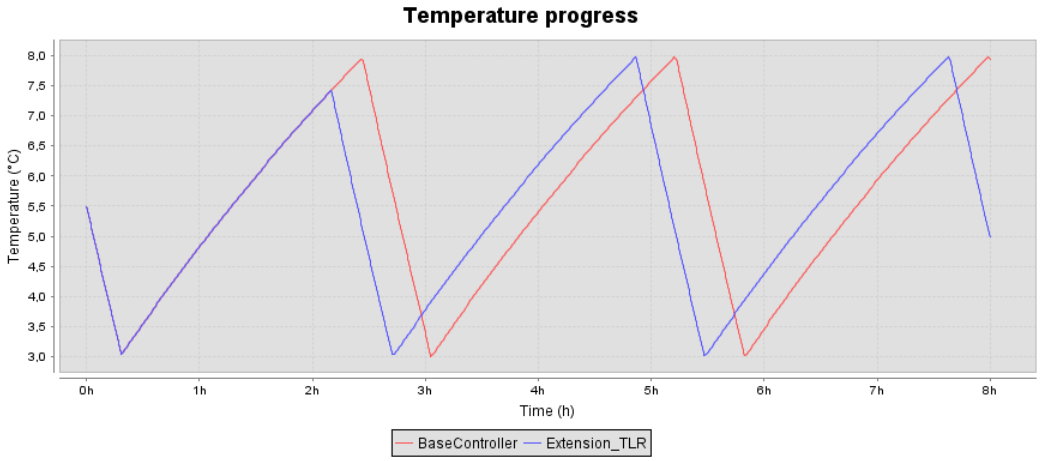


Abbildung C.3: Temperaturverlaufssimulationen der FSM BaseController und Extension_TLR mit $t_{notify} = 30\,min,\ \tau_{preload} = 10\,min,\ \tau_{reduce} = 1\,h\,30\,min$.

Es verbleiben noch die zwei möglichen Klassifizierungen C' und B', welche sich beide auf den Fall beziehen, dass das geforderte Reduktionsintervall größer ist als die mögliche Aufwärmdauer $\tau_{warming}$ des Gerätes. Abbildung C.4 zeigt die Klassifizierung C'. Hier war die Temperatur des Gerätes zum Zeitpunkt $t_{notify} = 1\,h$ niedrig genug,

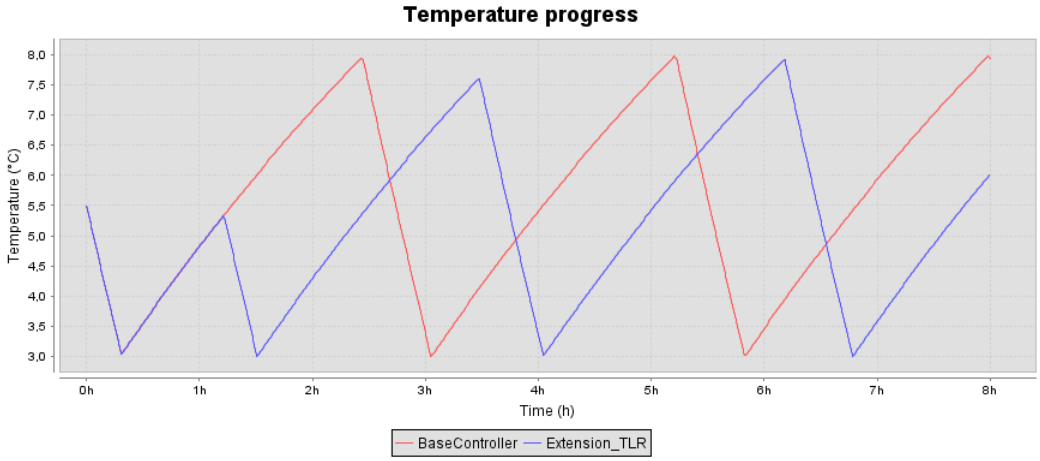


Abbildung C.4: Temperaturverlaufssimulationen der FSM BaseController und Extension_TLR mit $t_{notify} = 1\,h,\ \tau_{preload} = 30\,min,\ \tau_{reduce} = 2\,h\,30\,min$.

um zum Beginn des Reduktionsintervalles die Minimaltemperatur T_{min} zu erreichen

und somit die Reduktion möglichst lange durchzustehen. Im Gegensatz dazu ist in Abbildung C.5 der Fall dargestellt, dass die Temperatur zum Zeitpunkt $t_{notify} = 1\,h$

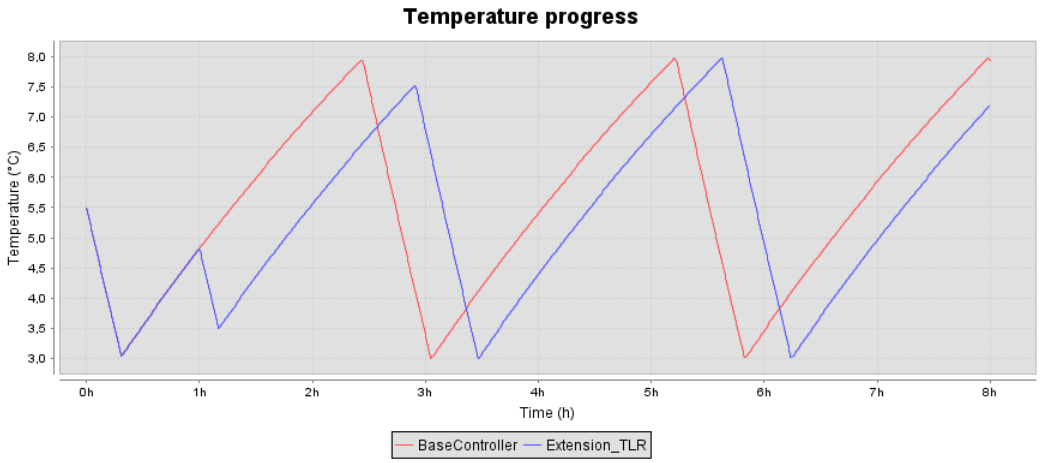


Abbildung C.5: Temperaturverlaufssimulationen der FSM BaseController und Extension_TLR mit $t_{notify} = 1\,h$, $\tau_{preload} = 10\,min$, $\tau_{reduce} = 2\,h\,30\,min$.

zu hoch war, um T_{min} zu erreichen. Diese Situation ähnelt der aus Abbildung C.2, da der Controller das ganze Intervall $\tau_{preload} = 10\,min$ zum Kühlen nutzt, um das Reduktionsintervall möglichst lange zu überstehen.

Anhang D

Vergleiche der DSC-*unload*-Simulationen

Die folgenden Diagramme zeigen jeweils Simulationsergebnisse des DSC-*unload*-Signales aus dem SimKit-Framework (blau) und aus dem FSM-Framework (rot). Die einzelnen Simulationen wurden mit unterschiedlichen Dämpfungsmechanismen durchgeführt (siehe jeweilige Bildunterschrift).

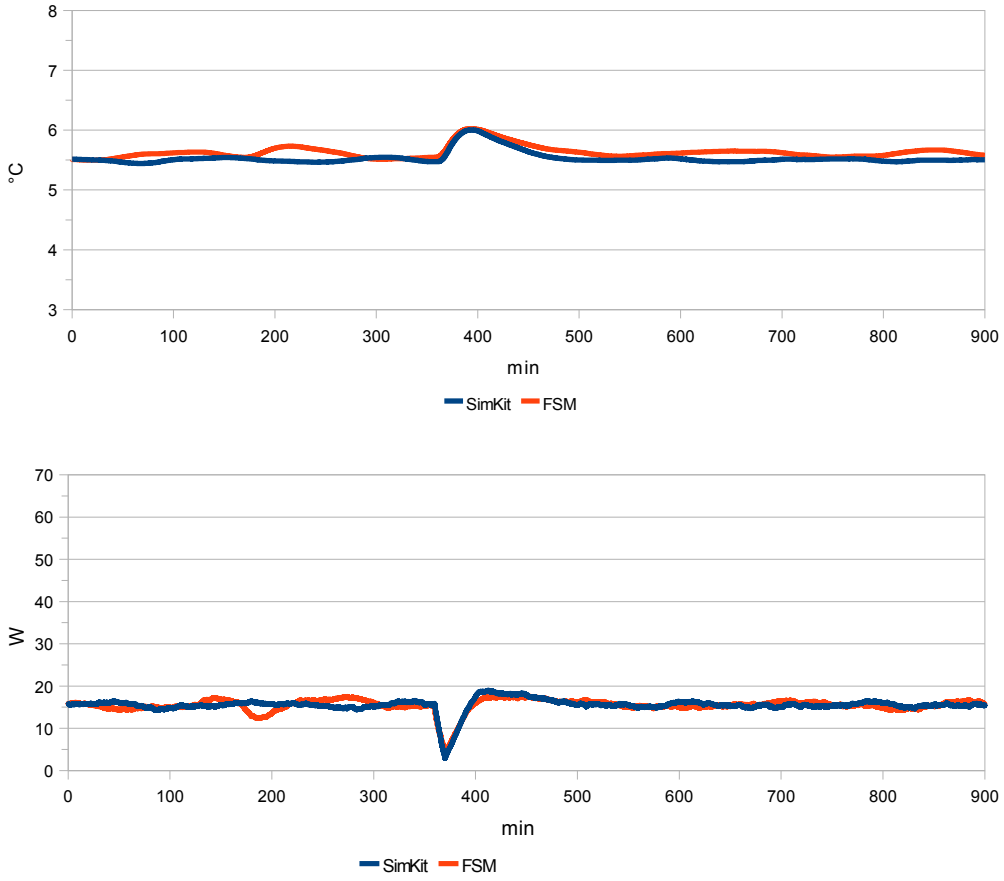


Abbildung D.1: Vergleich der SimKit- und FSM-Simulation mit DSC-*unload*-Signal und zustandsbasierter Dämpfung (*half-width*).

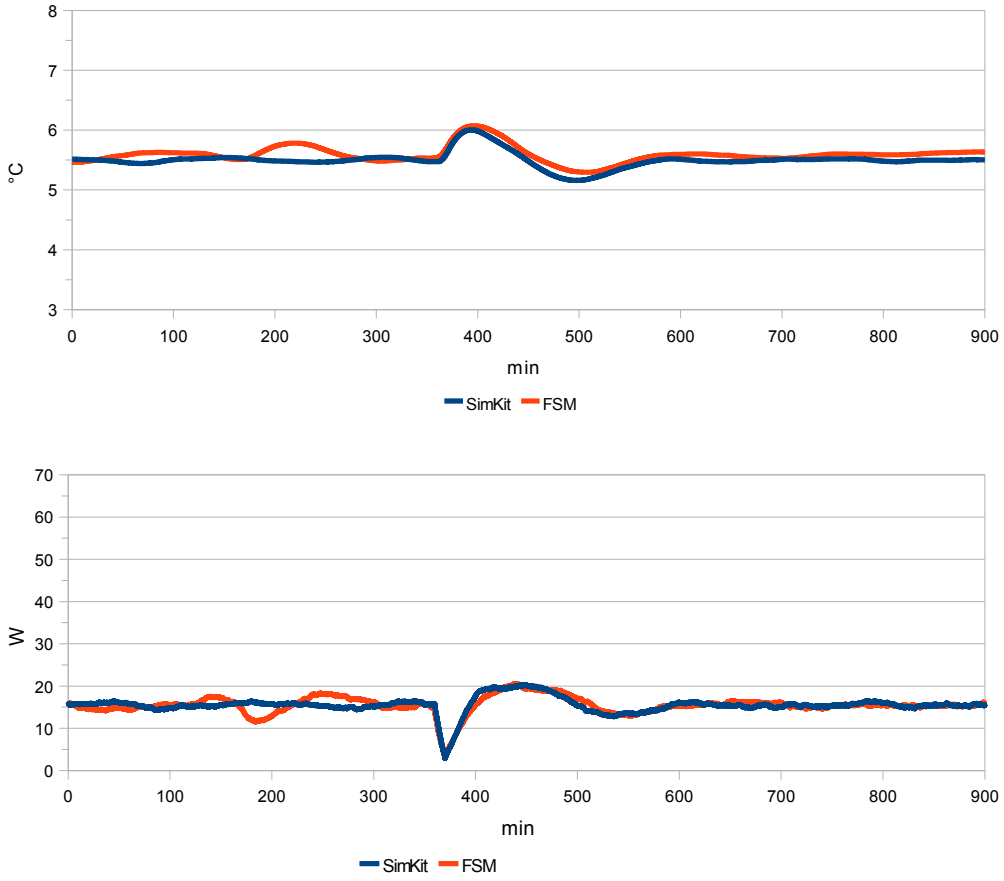


Abbildung D.2: Vergleich der SimKit- und FSM-Simulation mit DSC-*unload*-Signal und zustandsbasierter Dämpfung (*full-width*).

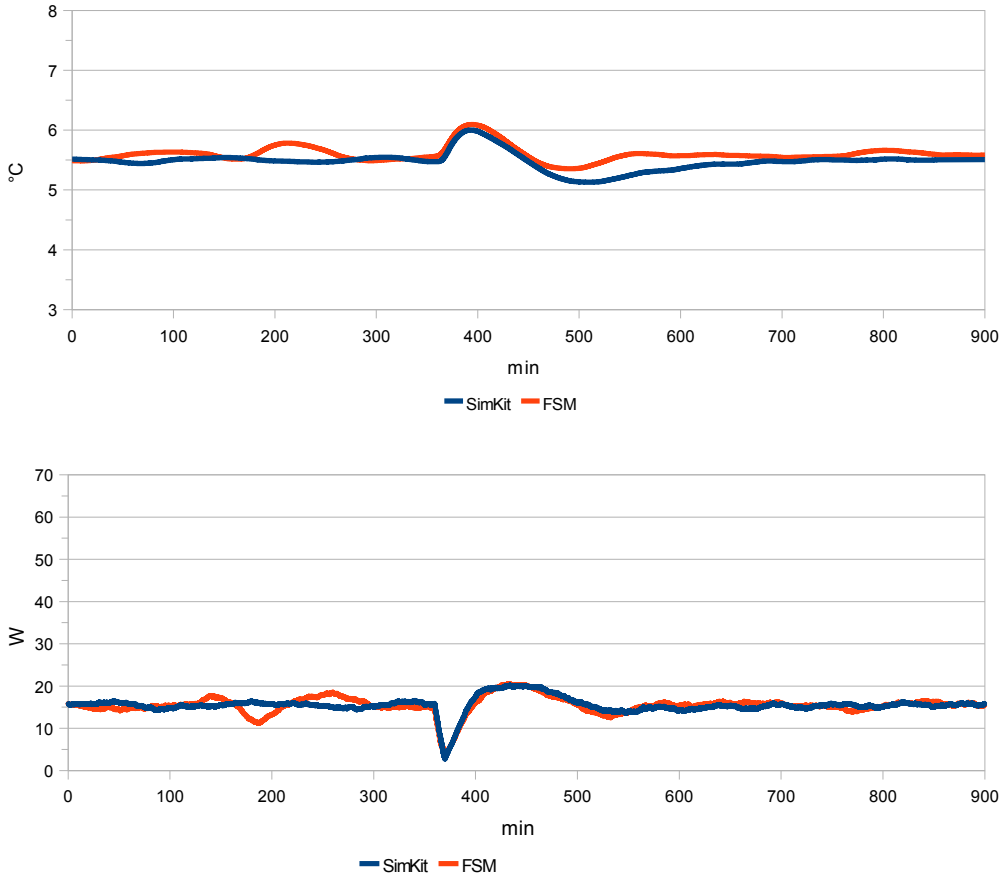


Abbildung D.3: Vergleich der SimKit- und FSM-Simulation mit DSC-*unload*-Signal und zufallsbasierter Dämpfung.

Anhang E

Inhalte der CD-ROM

File System: Joliet

Mode: Single-Session (CD-ROM)

Pfad: /

/Materialien/	Weitere nützliche Materialien zum Thema
/Projekte/	Entstandene Softwareprojekte, siehe unten
/Referenzen/	Referenzierte Ressourcen, aus dem Internet kopiert
_MSc.pdf	Diese Masterarbeit (PDF-File)
Readme.txt	Beschreibung der Inhalte der CDROM

Pfad: /Projekte

jfreechart-1.0.10-fastpatch/	Quellen der erweiterten JFreeChart Bibliothek als Eclipse 3.4 Projekt
Fridge/	Quellen der ereignisbasierten Simulationssoftware als Eclipse 3.4 Projekt
Fridge-webstart/	'Fridge' sowie Bibliotheken kompiliert für Java Web Start
FSM/	Quellen der FSM Simulationssoftware als Eclipse 3.4 Projekt

Literaturverzeichnis

[1] Apt, K. R. und. E. R. Olderog: *Programmverifikation - Sequentielle, parallele und verteilte Programme*. Springer-Verlag Berlin Heidelberg New York, 1991.

[2] Banks, J., J. Carson, B. L. Nelson und. D. Nicol: *Discrete-Event System Simulation*. Prentice Hall, 4 Aufl., 2004.

[3] Bourouis, A. und. B. Belattar: *JAPROSIM: A Java framework for Process Interaction Discrete Event Simulation*. Journal of Object Technology, 7(1):103–119, Januar 2008.

[4] Buss, A. H.: *Discrete Event Simulation On The World Wide Web Using Java*. Proceedings of the 1996 Winter Simulation Conference, S. 780–785, 1996.

[5] Buss, A. H.: *Basic Event Graph Modeling*. Simulation News Europe, 31, 2001.

[6] Buss, A. H.: *Discrete Event Programming with Simkit*. Simulation News Europe, 32/33, 2001.

[7] Buss, A. H.: *Component based simulation modeling with Simkit*. In:. *Simulation Conference, 2002. Proceedings of the Winter*, Bd. 1, S. 243 – 249, 2002.

[8] Buss, A. H.: *Composability and component-based discrete event simulation*. In:. *WSC '07: Proceedings of the 39th conference on Winter simulation: 40 years! The best is yet to come*, S. 694–702, 2007.

[9] Buss, A. H. und. P. J. Sánchez: *Modeling very large scale systems: building complex models with LEGOs (Listener Event Graph Objects)*. In:. *WSC '02: Proceedings of the 34th conference on Winter simulation*, S. 732–737. Winter Simulation Conference, 2002.

[10] Codl, D., J. Kaĉer und. T. Koutný: *Comparison-Evaluation of Java-Based Discrete-Time Simulation Tools*. 37th International Conference MOSIS-2003: Modelling and Simulation of Systems, 2003.

[11] Constantopoulos, P., F. C. Schweppe und. R. C. Larson: *ESTIA: a real-time consumer control scheme for space conditioning usage under sport electricity pricing*. Computers & Operations Research, 18(8):751–765, 1991.

[12] Douglas, B. P.: *UML Statecharts*. Embedded.com – The Official Site of the Embedded Development Community, http://www.embedded.com/1999/9901/9901feat1.htm, 2003.

[13] ETSO Balance Management Task Force: *Current State of Balance Management in Europe*. Techn. Ber., Europe Transmission System Operators, Brussels, Belgium, Dezember 2003.

[14] Gehlsen, B. und. B. Page: *A framework for distributed simulation optimization*. In:. *WSC '01: Proceedings of the 33nd conference on Winter simulation*, S. 508–514, Washington, DC, USA, 2001. IEEE Computer Society.

[15] Gellings, C. und. J. Chamberlin: *Demand-Side Management: Concepts And Methods*. The Fairmont Press, Inc., 1988.

[16] Graps, A.: *An Introduction to Wavelets*. IEEE Computational Science and Engineering, 2(2):50–61, 1995.

[17] Harel, D.: *Statecharts: A visual formalism for complex systems*. Sci. Comput. Program., 8(3):231–274, 1987.

[18] Hars, L. und. G. Petruska: *Pseudorandom Recursions: Small and Fast Pseudorandom Number Generators for Embedded Applications*. EURASIP Journal on Embedded Systems, vol. 2007, 2007.

[19] Helsgaun, K.: *Discrete Event Simulation in Java*. DATALOGISKE SKRIFTER (Writings on Computer Science), 2000.

[20] Helsgaun, K.: *jDisco - a Java framework for combined discrete and continuous simulation*. DATALOGISKE SKRIFTER (Writings on Computer Science), 2001.

[21] Howell, F. und. R. McNab: *simjava: A discrete event simulation library for java*. 1998 International Conference on Web-Based Modeling and Simulation, Januar 1998.

[22] Ibbett, R. N., P. E. Heywood und. F. W. Howell: *HASE: A Flexible Toolset for Computer Architects*. The Computer Journal, 38(10):755–764, 1995.

[23] Kaĉer, J.: *Discrete Event Simulations with J-Sim*. In:. Power, J. F. und. J. T. Waldron (Hrsg.): *Proceedings of the Inaugural Conference on the Principles and Practice of Programming in Java (PPPJ-2002)*, S. 13–18, Maynooth, Co. Kildare, Ireland, 2002. Department of Computer Science, National University of Ireland.

[24] Kähler, W.-M.: *Statistische Datenanalyse*. Friedr. Vieweg & Sohn Verlag/GWV Fachverlage GmbH, 3 Aufl., Oktober 2004.

[25] Knauer, U.: *Diskrete Strukturen - kurz gefasst*. Heidelberg: Spektrum, Akad. Verl., 2001.

[26] Law, A. und. W. D. Kelton: *Simulation Modeling and Analysis*. Mcgraw Hill Series in Industrial Engineering and Management Science. McGraw-Hill Inc., US, 2 Aufl., 1991.

[27] Lhotka, L.: *Implementation of individual-oriented models in aquatic ecology*. Ecological Modelling, 74:47–62, Juli 1994.

[28] Little, M. C.: *JavaSim User's Guide*. Department of Computing Science, Computing Laboratory, University of Newcastle upon Tyne, 1999.

[29] Miller, J. A., Y. Ge und. J. Tao: *Component-based simulation environments: JSIM as a case study using Java beans*. WSC '98: Proceedings of the 30th conference on Winter simulation, S. 373–382, 1998.

[30] Schruben, L.: *Simulation Modeling with Event Graphs*. Communications of the ACM, 26(11):957–963, November 1983.

[31] Shrivastava, S. K., G. N. Dixon und. G. D. Parrington: *An Overview of the Arjuna Distributed Programming System*. IEEE Software, 8(1):66–73, 1991.

[32] Sonnenschein, M. und. U. Vogel: *Modellbildung und Simulation ökologischer Systeme (Vorlesungsskript)*. Abt. Umweltinformatik, Dep. für Informatik, Carl von Ossietzky Universität Oldenburg, 2006.

[33] Stadler, I.: *Demand Response*. Doktorarbeit, Universität Kassel, 2006.

[34] Stadler, M., W. Krause, M. Sonnenschein und. U. Vogel: *The Adaptive Fridge - Comparing different control schemes for enhancing load shifting of electricity demand*. Hryniewicz, O., Studzinski, J., Romaniuk, M. (Eds.): 21st Conference Informatics for Environmental Protection - Enviroinfo Warsaw 2007, S. 199–206, 2007.

[35] Stork, K.: *Sensors In Object Oriented Discrete Event Simulation*. Diplomarbeit, MOVES Institute, Naval Postgraduate School, Monterey, California, 1996.

[36] Vangheluwe, H.: *Discrete Event Modelling and Simulation*. McGill University, School of Computer Science, http://www.cs.mcgill.ca/~hv/classes/MS/discreteEvent.pdf, 2001.

[37] Vignaux, T. und. K. Müller: *The SimpPy Manual*. http://simpy.sourceforge.net, März 2008.

[38] Vogel, U.: *Intelligentere Kontrollstrategien*. Vorläufiges Manuskript, unveröffentlicht, Mai 2008.

[39] Weber, K. und. W. Zillmer (Hrsg.): *Theoria Cum Praxi, TCP, Mathematik, Grundkurs*. Paetec, Berlin, 1 Aufl., 1995.

[40] Zeitz, M.: *Statistische Ergebnis-Auswertung für eine ereignis-diskrete Simulation*. Universität Stuttgard, ISR, 2001.