Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Florian Meyer

Implementierung einer adaptiven Entwicklungsumgebung zur Simulation von atomaren Manipulationen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Florian Meyer

Implementierung einer adaptiven Entwicklungsumgebung zur Simulation von atomaren Manipulationen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Schmidt
Zweitgutachter: PD Dr. Elena Vedmedenko

Eingereicht am: 9. Juli 2014

Florian Meyer

Thema der Arbeit

Implementierung einer adaptiven Entwicklungsumgebung zur Simulation von atomaren Manipulationen

Stichworte

Atommanipulation, Nanometer-Skala, Simulation, RCP, Modellierung, Messung, Monte-Carlo Simulation, Tunnelstrom

Kurzzusammenfassung

Unter Verwendung von Rastertunnelmikroskopen können Strukturen auf Nanometer-Skala künstlich erzeugt und verändert werden. Für diese Technik werden Kenntnisse der zugrundeliegenden Wechselwirkungen zwischen Atomen benötigt. Häufig werden Simulationen verwendet, die experimentelle Versuche nachahmen und so einen Erkenntnisgewinn begünstigen. Diese Arbeit befasst sich mit der Realisierung einer Software zur Durchführung von Simulationsstudien atomarer Strukturen. Dabei wird versucht die Komplexität der Anwendung durch eine grafische Benutzeroberfläche zu reduzieren. Um zukünftig entdeckte Verfahren und Techniken einsetzen zu können, wird der Ansatz der Erweiterbarkeit verfolgt. So kann die Software auf für zukünftige Erkenntnisse der Forschung verwendet werden.

Florian Meyer

Title of the paper

Implementation of an adaptive development environment for the simulation of atomic manipulation

Keywords

Atom manipulation, Nanometer-scale, Simulation, RCP, Modeling, Measurement, Monte Carlo simulation, Tunneling current

Abstract

Using scanning tunneling microscopes structures at nanometer-scale can be artificially created and modified. Knowledge of the underlying interactions are required for this technique. Often simulations are used to mimic the experimental trials and thus promote a gain in knowledge. This work deals with the implementation of a software performing simulation studies with atomic structures. The complexity of the Software is trying to reduced by a graphical userinterface. The approach of extensibility is tracked so this software can also be used with simulationmethods and techniques, that are discovered in the future.

Inhaltsverzeichnis

1 Einführung **1**

1.1 Motivation . . . 1
1.2 Problemstellung und Zielsetzung . . . 1
1.3 Struktur der Arbeit . . . 2

2 Grundlagen **3**

2.1 Simulation . . . 3
2.1.1 Grundlegende Begriffe . . . 3
2.1.2 Ablauf einer Simulationsstudie . . . 4
2.1.3 Komponenten einer Simulation . . . 6
2.1.4 Klassifizierung von Simulationsmodellen . . . 7
2.2 Manipulation atomarer Strukturen . . . 8
2.2.1 Rastertunnelmikroskopie . . . 8
2.2.2 Kristallstrukturen . . . 9
2.2.3 Wechselwirkungen . . . 10
2.2.4 Manipulation . . . 11
2.3 Rich-Client-Platform . . . 12
2.3.1 Client-Architekturen . . . 12
2.3.2 Eclipse Rich-Client-Platform . . . 13
2.3.3 Model-View-Control Pattern . . . 14

3 Konzept und Architektur **15**

3.1 Anforderungen . . . 15
3.1.1 Modellbildungsprozess . . . 15
3.1.2 Simulation . . . 16
3.1.3 Entwicklungsumgebung . . . 16
3.2 Abgrenzung . . . 17
3.2.1 Mögliche Einsatzszenarien . . . 17
3.2.2 Arbeitsablauf einer Simulationsstudie . . . 17
3.2.3 Verwendete Techniken . . . 19
3.3 Architektur der Komponenten . . . 20
3.3.1 Modellierung des Systems . . . 20
3.3.2 Prozessabbildung . . . 24
3.3.3 Simulationsausführung . . . 28
3.3.4 Analyse des Systems und Simulationsüberwachung . . . 29

3.3.5 Dokumentation der Ergebnisse und Auswertung 30

4 Realisierung **32**
4.1 Verwendete Bibliotheken 32
4.2 Realisierung der Entwicklungsumgebung 33
4.2.1 Plug-Ins 33
4.2.2 Model 34
4.2.3 View 38
4.2.4 Control 41
4.2.5 View-Model Interaktion 49
4.2.6 State Recovery 50
4.3 Realisierung der Komponenten 53
4.3.1 Platform 53
4.3.2 Environment 54
4.3.3 Simulations 56
4.3.4 Distributor 59
4.3.5 Monitor 60
4.3.6 Analysis 61

5 Test und Validierung **63**
5.1 Testen der Simulation 63
5.2 Verifikation des Simulationsprozesses 65

6 Zusammenfassung und Fazit **70**
6.1 Zusammenfassung der Zeile und Ergebnisse 70
6.2 Ausblick auf weitere Entwicklungen 71

A Ausschnitte der Entwicklungumgebung **73**

Literaturverzeichnis **77**

Glossar **80**

Abkürzungsverzeichnis **82**

Abbildungsverzeichnis **83**

Quellcodeverzeichnis **84**

1 Einführung

Mithilfe von Rastertunnelmikroskopen können leitende Festkörperoberflächen im Bereich weniger Nanometer untersucht und verändert werden. Diese Technik ermöglicht es Oberflächen mit atomarer Auflösung aufzunehmen sowie die magnetische Ausrichtung einzelner Atome zu beobachten. Dadurch können herkömmliche Speichermechanismen auf eine Größe von 12 Atomen pro Bit reduziert werden. Übliche Festplatten arbeiten mit ca. einer Million Atomen pro Bit. Darüber hinaus können logische Gatter aus wenigen Atomen konstruiert werden. Dieses hat nicht nur eine geringere Größe, sondern arbeitet wesentlich schneller und energiesparender als herkömmliche Silizium CMOS Technologien (vgl. Khajetoorians u. a. (2012)).

1.1 Motivation

Bis ein Rastertunnelmikroskop einsatzbereit ist und ein Experiment mit diesem durchgeführt werden kann, vergehen oft mehrere Tage. Diese Verzögerung resultiert aus dem komplexen Aufbau des Mikroskops und den hohen Anforderungen an die experimentelle Probe. Viele Untersuchungen werden bei einer Temperatur von nur wenigen Kelvin durchgeführt. Daher müssen die Mikroskope über Tage abgekühlt werden. Erst im Anschluss lässt sich feststellen, ob die Probe den Vorstellungen entspricht und alle Komponenten des Rastertunnelmikroskops betriebsbereit sind. Die Simulationstechnik bietet dabei eine wesentliche Optimierung. Durch diese kann ein reales Experiment zeitsparender untersucht werden, um Erwartungswerte zu liefern, die über Relevanz des Experiments mitentscheiden. Auch im Nachhinein kann eine Simulation durchgeführt werden, um zusätzliche Interpretationsansätze der physikalischen Vorgänge auf Nanometer-Skala zu liefern.

1.2 Problemstellung und Zielsetzung

Wünschenswert ist eine Softwareumgebung, mit der eine komplette Simulationsstudie durchgeführt werden kann. Oftmals sind Simulationsprogramme nur als Programmcode vorhanden. Dies ermöglicht eine schnellere Anpassung an neue Experimente, Techniken und Anforderungen. Dagegen benötigen komplexe Programmcodes für die Benutzer und Entwickler eine

längere Einarbeitungszeit und verringern die Übersicht. Daraus resultierende Fehler bei der Modellierung von Probe und Versuchsaufbau führen in den Simulationsergebnissen zu großen Abweichungen. Bei vielen Simulationen auf Nanometer-Skala sind oftmals nicht alle Parameter vorab einzuschätzen. Demzufolge müssen Simulationen mehrmals durchgeführt werden. Oftmals werden die Simulationen direkt im Programmcode parametrisiert, wodurch die Parameterwerte der Simulationsdurchläufe nicht persistent gespeichert werden oder nur durch Kopien des Codes vorhanden sind. Um diesen Nachteilen entgegen zu wirken, soll in der vorliegenden Arbeit eine übersichtliche und benutzerfreundliche Entwicklungsumgebung realisiert werden. Diese soll einem Anwender bei der Modellierung sowie der Simulationsdurchführung Validierungsmöglichkeiten bereitstellen. Damit die Software der Vielseitigkeit eines Rastertunnelmikroskops genügt, müssen Erweiterungsmöglichkeiten vorgesehen werden. Des Weiteren soll die Simulationsdurchführung möglichst performant sein, um einen hohen Simulationsdurchsatz zu erzielen. Daher wird das Programm so konzipiert, dass eine verteilte Berechnung auf mehreren Prozessorkernen ermöglicht wird.

1.3 Struktur der Arbeit

Die vorliegende Arbeit gliedert sich wie folgt: In Kapitel 2 werden grundlegende Begriffe sowie das typische Vorgehen von Simulationsstudien beschrieben. Weiterhin werden Begriffe der atomaren Manipulation sowie die experimentelle Untersuchung mit Rastertunnelmikroskopen beschrieben. Zuletzt wird eine Einführung in die verwendete Rich-Client-Platform und dessen Architektur gegeben, mit welcher die Software modularisiert wird. In dem Zusammenhang wird das MVC-Pattern beschrieben, welches bei der Realisierung eingesetzt wurde. Kapitel 3 stellt zunächst die Anforderungen des Modellbildungsprozesses, der Simulation sowie der Entwicklungsumgebung dar. Im Weiteren wird beschrieben, in welchem Rahmen diese Arbeit entwickelt ist und wo die möglichen Einsatzgebiete liegen. Zuletzt wird das entworfene Konzept und dessen Techniken vorgestellt. Damit wird ein Überblick über die Architektur der Entwicklungsumgebung und die Einteilungen der einzelnen Komponenten gegeben. Kapitel 4 zeigt die Umsetzung der Entwicklungsumgebung und der Komponenten, die den Arbeitsablauf einer Simulationsstudie realisieren. Dazu werden zuerst die verwendeten Software-Bibliotheken vorgestellt und anschließend die Umsetzung der wichtigsten Techniken und Konzepte erläutert. Kapitel 5 stellt die durchgeführten Test der Hauptkomponente zur Simulation vor. Darüber hinaus wird das Validierungsverfahren der Entwicklungsumgebung und dessen Ergebnisse beschrieben. Abschließend enthält Kapitel 6 eine Zusammenfassung der Arbeit und gibt einen Ausblick auf weitere Entwicklungen in diesem Zusammenhang.

2 Grundlagen

In diesem Kapitel werden grundlegende Begriffe sowie verwendete Techniken erklärt. In Abschnitt 2.1 wird eine Einführung in die Thematik der Simulationstechnik gegeben. Abschnitt 2.2 beschreibt die verwendete Technik zur Atommanipulation und erläutert damit zusammenhängende Begriffe. Zuletzt wird in Abschnitt 2.3 die verwendete Rich-Cline-Platform vorgestellt, mit der die Entwicklungsumgebung realisiert wird.

2.1 Simulation

Simulationen sind Analysemethoden, die in vielen Gebieten zum Einsatz kommen. Abläufe eines realen Systems können anhand eines vereinfachten Modells untersucht werden. Erkenntnisse, die dabei gewonnen werden, lassen Rückschlüsse auf das Verhalten des realen Systems zu (vgl. Page und Häuslein (1991)). Eine wichtige Anwendung der Simulation ist das Analysieren von Vorgängen, die in der Realität zu schnell oder zu langsam ablaufen und sich dadurch einer praktikablen Untersuchung entziehen. Einen weiteren Vorteil bietet die Simulation von Prozessen, deren Durchführung in der Realität zu teuer wären. Auch das approximative Vorhersagen von realen Systemen, wodurch die Simulation als Entscheidungshilfe interessant wird, ist eines der vielseitigen Einsatzbereiche der Simulationstechnik (vgl. Neelamkavil (1987)).

2.1.1 Grundlegende Begriffe

Bei Simulationen sind zunächst folgende Begrifflichkeiten zu erläutern, um den Zusammenhand zu betrachten. Diese wurden nach Kramer und Neculau (1998) wie folgt definiert:

Prozess Jedes Phänomen, das zeitlich-räumlichen Veränderungen unterworfen ist, kann als ein Prozess aufgefasst werden. Er ist dadurch gekennzeichnet, dass es mindestens eine Größe gibt, die sich mit voranschreitender Zeit und/oder in Abhängigkeit vom Ort verändert. Wenn die interessierenden Größen $x,y,etc.$ von den Parametern Zeit t und Ort r abhängen, werden diese Größen als Prozessvariablen $x(t,r)$, $y(t,r)$,etc. bezeichnet.

System Ein System ist die gedankliche Abgrenzung einer realen Gegebenheit, die zugleich festlegt, was als Systeminneres, was als Systemumgebung und was als Systemgrenze anzusehen ist. Durch die Abgrenzung werden zugleich Größen definiert, die im Zusammenhang mit einer konkreten Aufgabenstellung von Interesse sind. Dabei unterscheidet man die Größen danach, ob sie sich in Abhängigkeit von Zeit- und/oder Raumparametern t bzw. r ändern. Im Falle einer Abhängigkeit spricht man von *Prozessvariablen*; Im Falle einer Unabhängigkeit werden die Größen als *Systemparameter* bezeichnet. Bei den Systemparametern unterscheidet man weiter zwischen *Eingangsgrößen* und den davon abhängigen *Ausgangs-* oder *Verhaltensgrößen*. Eingangsgrößen, die Änderungen der Ausgangsgrößen bewirken, ohne definierbar zu sein und häufig nicht einmal messbar sind, werden als *Störgrößen* definiert. All diese Größen bilden den Zusammenhang, der die Struktur eines Systems kennzeichnet.

Modell Ein Modell ist die formale Nachbildung der Abhängigkeiten einer oder mehrerer Prozessvariablen $x(t,r)$, $y(t,r)$,etc. von der Zeit t und den Ortskoordinaten r. Darüber hinaus bildet ein Modell auch den Zusammenhang zwischen den einzelnen Prozessvariablen nach. Jedes Modell ist ein abstraktes System, bei dem zwischen Eingangs- und Ausgangsgrößen, Steuer- und Störgrößen sowie Systemparametern zu unterscheiden ist.

Simulation Eine Simulation ist die Nachbildung realer Prozesse mittels Computern auf der Grundlage eines Modells.

2.1.2 Ablauf einer Simulationsstudie

Zur erfolgreichen Durchführung einer Simulationsstudie wird die Vorgehensweise bei Simulationsprojekten in mehrere Durchführungsphasen (nach Warschat und Wagner (1995)) untergliedert. Nachfolgend werden die Phasen in Reihenfolge ihrer Abarbeitung beschrieben (siehe Abbildung 2.1).

Problemdefinition Zu Beginn jeder Simulationsstudie muss das genaue Ziel der Untersuchung festgelegt werden. Dabei definiert man eine Abgrenzung des Problems durch Festlegung des Systeminneren, der Systemumgebung und der Systemgrenzen.

Analyse der Systemparameter und Daten Zuerst müssen alle relevanten Systemparameter (Eingangs- und Ausgangsgrößen) ermittelt werden, um eine ausreichende Validität des Modells zu erreichen. Nachdem die wesentlichen Systemparameter bestimmt sind, ist festzulegen, welche Daten für diese Parameter notwendig sind.

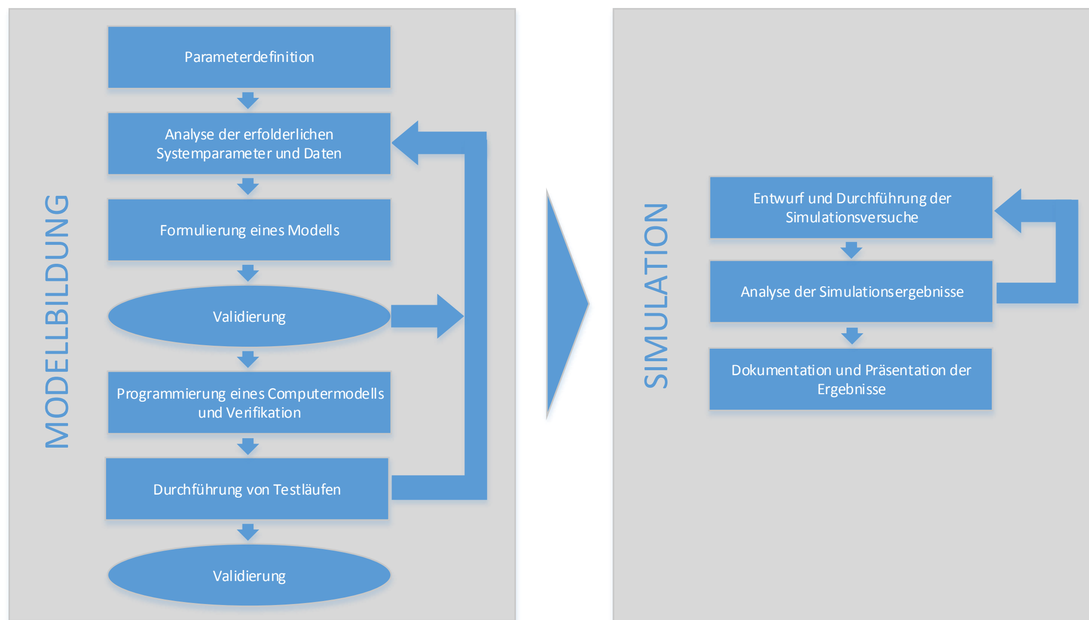

Abbildung 2.1: Vorgehensmodell in Simulationsstudien (angelehnt an Warschat und Wagner (1995))

Formulierung eines Modells Anfangs sollte mit einem möglichst groben Modell begonnen werden, das aus mathematischen oder logischen Regeln besteht. Dabei sollten nur die Systemparameter abgebildet werden, die als wichtig erachtet sind. Im Laufe der Simulationsstudie kann das Modell nach und nach detaillierter beschrieben werden.

Validierung des formulierten Modells Um ein Modell auf Übereinstimmung mit seinem Realsystem und der Aussagefähigkeit gewonnener Ergebnisse zu überprüfen, sollten zwei Aspekte der Modellierung genauer betrachtet werden. Zum einen der richtige Aufbau der Objekte und dessen Abhängigkeit untereinander, sowie zum andern die naturgetreue Verteilung der auftretenden stochastischen Einflüsse.

Modellierung eines Computermodells und Verifikation Nach der Formulierung eines mathematischen bzw. logischen Modells, das den gegebenen Rahmenbedingungen entspricht, wird das Modell in Form von Programmcode implementiert. Dazu können entweder Programmiersprachen oder Simulationswerkzeuge verwendet werden. Nach der Implementierung wird auch dieses Modell auf Validität überprüft. Eine mögliche Methode bietet die Simplifizierung: Simulationsläufe mit vereinfachten Bedingungen durchführen, um dadurch die Ergebnisse

voraussagbar und damit verifizierbar zu machen. Darüber hinaus gibt es noch weitere Möglichkeiten der Verifizierung, die hier nicht weiter thematisiert werden.

Durchführung von Testläufen Um das Computermodell besser zu validieren, werden Testläufe durchgeführt, die vor allem die Sensitivität der Systemparameter untersuchen. Sollten geringe Änderungen in den Eingangsgrößen zu unverhältnismäßigen Änderungen der Ausgangsgrößen führen, muss ggf. der Wertebereich der Eingangsgrößen überprüft werden. Eine Abschätzung der Systemparameter kann durch Erkenntnisse aus ähnlichen Modellen erfolgen, so lässt sich eine höhere Validität des Modells erreichen.

Entwurf und Durchführung der Simulationsversuche Bei der Planung von Prozessen entstehen oftmals mehrere Systemalternativen. Da die Anzahl der möglichen Versuche exponentiell mit der Zahl der zu untersuchenden Systemparameter und deren Varianten ansteigt, muss entschieden werden, welche dieser Alternativen für die Simulationsstudie in Betracht gezogen werden sollten.

Analyse der Simulationsergebnisse Die Aussagefähigkeit der Ausgangsdaten und ihre statistische Verwertbarkeit muss überprüft werden, womit auch das Modell erneut validiert werden kann. Anhand der Simulationsergebnisse kann bei richtiger Beurteilung eine Entscheidung für oder gegen die Durchführung eines Projektes getroffen werden.

Dokumentation und Präsentation der Ergebnisse Eine Simulationsstudie ist erst nach der Dokumentation der Ergebnisse vollständig abgeschlossen. Hierbei ist es nicht nur wichtig alle Ergebnisse zu dokumentieren, sondern auch die Erkenntnisse bei der Modellierung, sowie aller Annahmen, die dem Modell zugrunde liegen. Dadurch wird eine Simulationsstudie für zukünftige Untersuchungen mit ähnlichen Modellen verwendbar.

2.1.3 Komponenten einer Simulation

Eine Simulation bildet sich aus mehreren Komponenten, deren Zusammenwirken die Durchführung von Versuchen ermöglicht.

Simulationsmonitor Bei vielen Simulationsprogrammen existiert ein Simulationsmonitor, in dem ein mehr oder weniger detaillierter Ablauf der Simulation dargestellt wird. Animierte Darstellungen müssen aus Performanzgründen während der Simulationsdurchführung oft

deaktiviert werden. Eine Visualisierung der Simulationsdurchführung ist in manchen Fällen nicht sinnvoll, wenn die Simulationszeit eine detaillierte Betrachtung der Vorgänge erschwert.

Modellstruktur Abgrenzung einer realen Gegebenheit, die das System repräsentiert. Typischerweise besteht das Modell aus mehreren Objekten, die miteinander interagieren und deren Zustand sich in Abhängigkeit der Simulationszeit verändert. Je nach Art und Weise, wie die simulierte Realzeit voranschreitet, lassen sich die Simulationsmodelle klassifizieren (siehe Abschnitt. 2.1.4).

Simulationsuhr Steuerung des zeitlichen Ablaufs der einzelnen Prozessschritte. Abhängig vom gewählten Simulationsmodell kann die Simulationszeit auf den Zeitpunkt des nächsten auftretenden Ereignisses gestellt werden oder um ein festgelegtes Zeitintervall erhöht werden.

Auswertung Die in der Simulation erzeugten Ergebnisse werden von dieser Komponente aufbereitet, interpretiert und ausgewertet. Je nach Anforderung werden die Daten in Diagrammen, Tabellen oder Animationen angezeigt (vgl. Warschat und Wagner (1995)).

2.1.4 Klassifizierung von Simulationsmodellen

Abhängig vom zeitlichen Verhalten eines realen Systems existieren unterschiedliche Simulationsmodelle. Anhand der simulierten Realzeit (im Folgenden Simulationszeit genannt) lassen sich die Simulationsmodelle wie folgt klassifizieren:

Bei **kontinuierlicher** Simulation schreitet die Simulationszeit kontinuierlich voran. Die meisten Modelle dieser Art bestehen aus Differentialgleichungssystemen mit zeitlich abhängigen Größen.

Diskrete Simulationen dagegen ändern ihren Zustand sprunghaft zu bestimmten Zeitpunkten (im folgenden Ereignis genannt). Die Eintrittszeit t eines Ereignisses wird oft auch als Zeitstempel bezeichnet. Je nach Steuerung des Simulationsablaufs durch die Simulationsuhr kann zwischen zeit- und ereignisgesteuerter Simulation unterschieden werden.

Bei **zeitgesteuerten** Simulationen erhöht sich die Simulationszeit t um feste oder variable Schrittlängen. Alle Ereignisse eines Intervalls werden in willkürlicher Reihenfolge ausgeführt und dürfen nur Ereignisse beeinflussen, die in zukünftigen Intervallen liegen. Andernfalls muss ein dementsprechend kleineres Intervall gewählt werden. Dies kann allerdings auch Totzeiten verursachen, falls das Intervall so klein gewählt wurde, dass sich stellenweise kein Ereignis darin befindet. Das Risiko, solche Totzeiten zu erzeugen, stellt einen wesentlichen Nachteil zeitgesteuerter Simulationen dar.

Die **ereignisgesteuerte** Simulation ist dadurch charakterisiert, dass die Simulationszeit bis zum nächsten auftretenden Ereignis erhöht wird. Dies bedeutet, dass Ereignisse die Simulationszeit steuern und keine Totzeiten entstehen. Ereignisse können neue Ereignisse in der Zukunft auslösen. Somit können auch komplexe Verhalten simuliert werden (vgl. Mehl (1994)).

2.2 Manipulation atomarer Strukturen

2.2.1 Rastertunnelmikroskopie

Funktionsweise Die Technik der Rastertunnelmikroskope (RTMs) wurde Anfang der 80er Jahre von Gerd Binning und Heinrich Roher im IBM-Forschungslabor Zürich entwickelt (vgl. Binnig und Rohrer (1987)). Das Funktionsprinzip eines RTMs basiert auf dem Effekt des quantenmechanischen Tunnelns (Tunneleffekt): Elektronen können eine Potenzialbarriere durchqueren, die sie aufgrund der klassisch-physikalischen Gesetze nicht überwinden könnten. Die Barriere besteht aus einer isolierenden Schicht aus Luft oder einem Vakuum zwischen zwei Elektroden. Die eine Elektrode stellt die Oberfläche der Probe dar, die andere eine feine Spitze, die in einem geringen Abstand über die Probe gerastert wird. Legt man zwischen beiden eine Spannung an, fließt ein Tunnelstrom, dessen Größe exponentiell vom Elektrodenabstand abhängt. Der Strom, aufgetragen über die x- und y-Position, gibt Auskunft über das Höhenprofil der Probe (vgl. Wautelet u. a. (2003)). Das RTM kann in diesem Zusammenhang in zwei Betriebsmodi betrieben werden (siehe Abbildung 2.2).

Betriebsmodi Im **Konstant-Strom-Modus** wird der Tunnelstrom durch eine Veränderung der Spitzenposition in Z-Richtung an jedem Messpunkt konstant gehalten. Das eigentliche Messsignal stellt in diesem Fall die Regelspannung der Spitze dar, welche die Topografie der Probenoberfläche widerspiegelt. Dieser Modus eignet sich besonders zur Auflösung sehr rauer Oberflächenstrukturen. Der **Konstant-Höhe-Modus** bietet den Vorteil einer höheren Abtastrate, da die Spitze in fester z-Position über die Oberfläche geführt wird und somit die Spitzenregelung entfällt. Die Änderung des Tunnelstroms wird dabei aufgezeichnet und dient als Maß für die lokale Leitfähigkeit. Auf diese Weise gewinnt man Informationen über die Oberflächenbeschaffenheit. Dieser Modus kann nur auf atomar glatten Proben eingesetzt werden, da andernfalls eine Kollision zwischen Spitze und Unebenheiten der Oberfläche eine Verformung beider Materialien zu Folge haben kann (vgl. University Ulm (2014)).

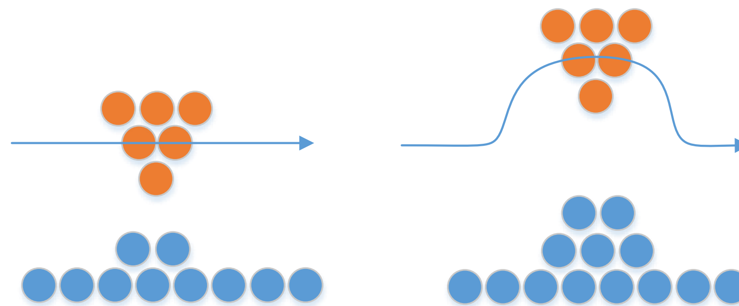

Abbildung 2.2: Konstant-Höhe-Modus (links) und Konstant-Strom-Modus (rechts)

2.2.2 Kristallstrukturen

In der Natur sind häufig bestimmte Gitterstrukturen vorhanden. Im Periodensystem der Elemente sind drei unterschiedliche Gittertypen zu finden (siehe Abbildung 2.3), die sich jeweils aus Atomen eines Elementtyps erstellen lassen: kubisch raumzentrierte Gitter, kubisch flächenzentrierte Gitter und hexagonal dichteste Kugelpackung. In dieser Arbeit wird ausschließlich mit diesen Gittertypen gearbeitet, sodass komplexere nicht thematisiert werden.

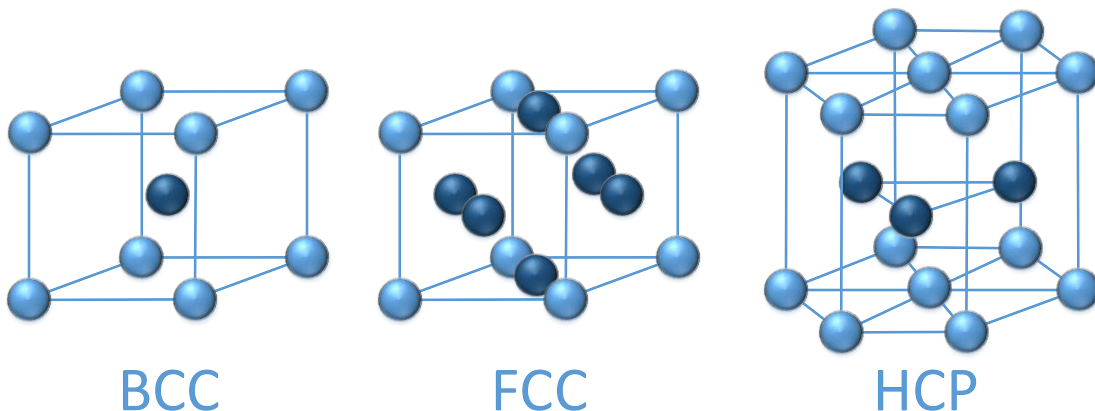

Abbildung 2.3: Häufig vorkommende Kristallgitter

Kubisch raumzentriert Eine sehr häufig vorkommende Gitterstruktur der Alkalimetalle sowie Eisen, Chrom, Barium und vielen anderen Metallen und Verbindungen. Oft abgekürzt mit *bcc* (englisch für body centered cubic).

Kubisch flächenzentriert Edelgaskristalle und viele Metalle wie Kupfer, Silber, Gold und Nickel besitzen ein kubisch flächenzentriertes Gitter. Abgekürzt mit *fcc* (englisch für face centered cubic).

Hexagonal dichteste Kugelpackung Viele Metalle wie Rhenium, Magnesium, Titan, Kobalt und viele weitere weisen diese Struktur auf. Der Abstand der nächsten Nachbaratome ist so groß wie beim kubisch flächenzentriert Gitter. Abgekürzt mit *hcp* (englisch hexagonal close packed) (vgl. Breu u. a. (2008)).

2.2.3 Wechselwirkungen

In dieser Arbeit werden zwei Wechselwirkungen verwendet, um die Energie eines Atoms in einem geschlossene System zu bestimmen. Die Algorithmen zur Berechnung dieser Teilenergien wurden aus einer Vorarbeit (siehe Wolter (2014) S. 53) übernommen und für die zu entwickelnde Anwendung angepasst.

Chemische Wechselwirkung Die chemische Interaktion zweier Atome wurde durch das Morse Potential[1] Modelliert. Die chemische Potential-energie U_{Morse} zweier Atome i und j mit einer Distanz r wird berechnet durch:

$$U_{Morse}(r) = U_0 \exp^{-2\alpha(r-r_0)} - 2U_0 \exp^{-\alpha(r-r_0)}$$

U_0, α und r_0 stellen die drei Parameter des Potentials dar.

Magnetische Wechselwirkung Die magnetische Interaktion zweier Atome wurde durch den Heisenberg Exchange Hamiltonian[2] modelliert. Die magnetische Energie J_i zweier Atome i und j mit einer Distanz r und einer magnetischen Ausrichtung S_i und S_j wird berechnet durch:

$$J_i(r) = J_0 \exp^{-\alpha(r-r_0)}$$

J_0, α und r_0 stellen die drei Parameter des Hamiltonian dar (vgl. Wolter (2014)).

[1] Beschreibt den Verlauf des elektronischen Potentials eines zweiatomigen Moleküls
[2] Der Hamiltonian dient in der Quantenmechanik zur Bestimmung von Energien eines Systems

2.2.4 Manipulation

Die Manipulation einzelner Atome ist eine innovative experimentelle Technik der Nano-Forschung. Dabei wird die Spitze eines RTM verwendet, um künstlich atomare Strukturen zu erzeugen, neuartige Quantenphänomene zu untersuchen und Eigenschaften einzelner Atome sowie Moleküle auf atomarer Ebene zu analysieren. Die RTM-Manipulation kann durch präzise Steuerung der Spitze-Proben-Wechselwirkung durchgeführt werden (vgl. Hla (2005))

Manipulation mit Adatom Ein RTM-Manipulationsprozess zum Bewegen einzelner Atome/Moleküle entlang einer Oberfläche nennt man Laterale Manipulation (LM). Das erste Beispiel für eine LM wurde im Jahr 1990 gezeigt. Dabei wurde auf einer Nickeloberfläche der Schriftzug IBM mit Xenon-Atomen geschrieben (vgl. Toumey (2010)). Zum Durchführen einer LM wird zunächst die Oberflächenstruktur aufgenommen, um ein freiliegendes Atom (Adatom) zu lokalisieren. Anschließend wird die Spitze des RTMs in einem geringen Abstand über das Adatom bewegt, um die Spitzen-Adatom-Wechselwirkung zu verstärken. Der Abstand ist hierbei geringer als beim üblichen Topografie-Prozess (siehe Abschnitt 2.2.1). Anschließend wird mit der Spitze eine laterale Bewegung entlang der Oberfläche durchgeführt, wobei sich das Adatom unter Einfluss der Spitze bewegt. Zuletzt wird ein weiteres Bild der Oberfläche aufgenommen, um die neue Position des Adatoms zu überprüfen (siehe Abbildung 2.4).

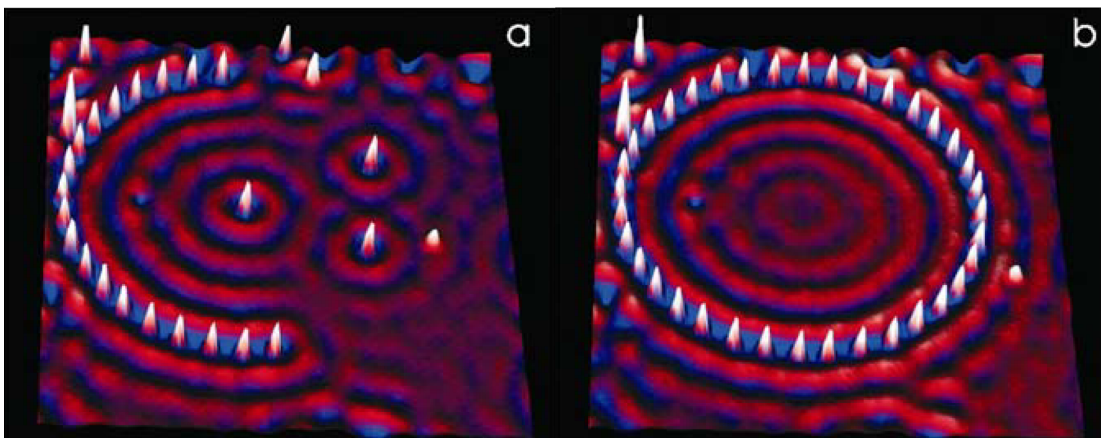

Abbildung 2.4: Dreidimensionales RTM-Bild. 36 Ag-Atome kreisförmig angeordnet (Quelle: Hla u. a. (2003))

2.3 Rich-Client-Platform

2.3.1 Client-Architekturen

In Folgendem werden die Funktionsweisen, sowie dessen Vor- und Nachteile der Fat- und Thin-Clinets beschrieben (vgl. Sommerville (2007)). Anschließend werden die beiden Architekturen der Rich-Client-Architektur gegenübergestellt.

Fat-Client Anwendungen übernehmen vollständig die Berechnung der Daten, deren Aufbereitung und die grafische Präsentation auf dem Client. Lediglich für Datenhaltung und Datenbankoperationen werden Server eingesetzt. Vorteile der Fat-Clients liegen in der hohen Performanz bei der Verarbeitung großer Datenmengen sowie die gute Benutzerführung bei der Verarbeitung. Durch die Offline-Fähigkeit wird eine geringe Abhängigkeit zwischen Server und Client realisiert. Nachteilhaft ist die Plattformabhängigkeit und der erhöhte Aufwand bei der Wartung und Bereitstellung der Anwendung.

Thin-Client Anwendungen werden lediglich zur grafischen Präsentation der Daten verwendet. Die Datenhaltung und Verarbeitung erfolgt ausschließlich auf einem Server. Vorteile gegenüber dem Fat-Client Ansatz sind die Plattformunabhängigkeit sowie ein geringerer Aufwand bei Wartung und Bereitstellung der Anwendung. Nachteile sind bei der Verarbeitung großer Datenmengen zu sehen, die von beschränkter Datentransferrate sowie Serverrechenkapazität abhängen. Zudem eignen sich diese Anwendungen nicht für einen Offline-Betrieb, da sie an eine permanente Serververbindung gebunden sind. Ein typisches Beispiel für Thin-Client Anwendungen sind Web-Browser, die lediglich HTML-Daten von einem Webserver präsentieren.

Rich-Client Anwendungen bieten eine qualitativ hochwertige Endbenutzerhandhabung sowie eine reichhaltige Funktionalität in der Benutzerinteraktion durch Bereitstellung vielfältiger, nativer Methoden[3] wie z.B. die System-Zwischenablage, Drag & Drop, Tastaturkürzel sowie viele Möglichkeiten zur Anpassung der Oberflächen an die Bedürfnisse der einzelnen Benutzer. Darüber hinaus übernehmen Rich-Client-Anwendungen die Vorteile der Fat-Clients große Datenmengen serverunabhängig zu verarbeiten. Gleichzeitig sind auch die Vorteile der Thin-Clients, wie die Plattformunabhängigkeit, der verbesserten Softwareverteilung und Wartung, bei den Rich-Client Anwendungen zu finden (vgl. McAffer u. a. (2010), Daum (2008)).

[3] Sourcecode der direkt vom Betriebsystem ausgeführt wird. Anders als Java Code werden native Methoden nicht in der virtuellen Maschine ausgeführt

2.3.2 Eclipse Rich-Client-Platform

Eclipse wurde bis zur Version 2.1 hauptsächlich als reine Entwicklungsumgebung für Java Anwendungen benutzt. Seit der Version 3.0 wurde mit einer überarbeiteten Architektur die Möglichkeit geschaffen, Rich-Client-Anwendungen zu entwickeln. Dabei bildet die Rich-Client-Platform (RCP) einen kleinen Kern, der dafür benötigt wird Plug-Ins auszuführen. Die ursprüngliche Eclipse-IDE ist jetzt eine spezielle Rich-Client-Anwendung, die auf der Eclipse RCP aufsetzt.

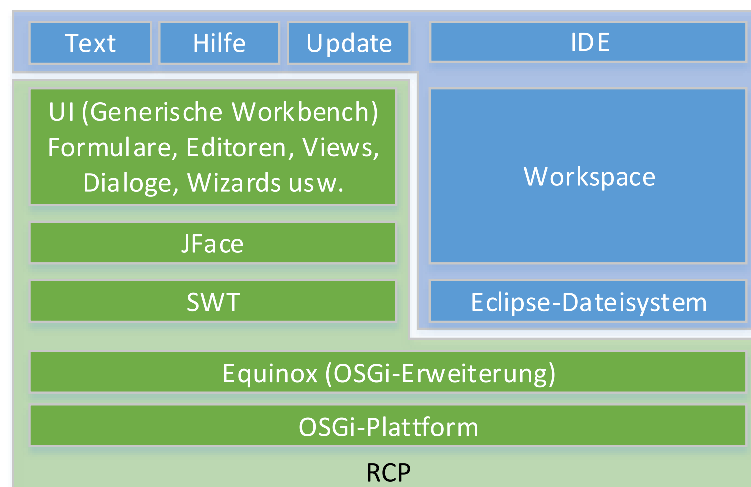

Abbildung 2.5: Die Eclipse Platform und dessen Komponenten (angelehnt an Daum (2008))

Der größte Teil der RCP besteht aus Komponenten für die Benutzeroberfläche (dargestellt als hellgrüner Bereich in Abbildung 2.5). OSGi spezifiziert eine Java-Laufzeitumgebung mit der die Ausführung von Modulen (Plug-Ins) ermöglicht wird. Diese Spezifikation wird von Eclipse als Equinox implementiert. Das Standard-Widget-Toolkit (SWT) und JFace stellen die Basisfunktionalität für grafische Oberflächen (GUI) bereit. Zur Darstellung der Benutzeroberfläche stellt SWT betriebssystemspezifische Aufrufe zur Verfügung, wodurch es äußerlich kaum von nativen Anwendungen zu unterscheiden ist. SWT wird für viele Betriebssysteme als Plug-In bereitgestellt, um Plattformunabhängigkeit zu ermöglichen. JFace stellt aus den SWT Komponenten komplexere Widgets[4] zusammen und regelt dessen komplette Verwaltung. So sind die Grundstrukturen für den Aufbau von GUI-Anwendungen durch JFace vorgegeben und müssen nicht erst entwickelt werden. Gleichzeitig entsteht ein einheitliches Bedienkonzept. Auf den GUI-Bibliotheken baut die generische Workbench auf, die u.a. Menü- und Werkzeu-

[4] Komponenten eines grafischen Fenstersystems, das den Zustand eines Objekts visualisiert

gleisten, Editoren, Views, Dialoge und Wizards verwaltet. Generisch bedeutet hierbei, dass diese Funktionalitäten optional in jede Rich-Client-Anwendung eingebunden werden können (vgl. Ebert (2011), Daum (2008)).

Plug-In Manifest Für die Erzeugung von Plug-Ins wird ein spezielles Plug-In Manifest erstellt. Dort werden wichtige Informationen für das Einbinden zur Laufzeit hinterlegt. So können Erweiterungen anderer Plug-Ins deklariert werden, ohne die konkrete Klassenreferenz im Plug-In zu enthalten. Weiterhin wird in dem Manifest deklariert, welche Schnittstellen für andere Plug-Ins nutzbar sein sollen. Auch die Abhängigkeiten eines Plug-Ins werden dort festgehalten. So kann beim Einbinden des Plug-Ins zur Laufzeit überprüft werden, ob die benötigten Plug-Ins bereits vorhanden sind. Das Auswerten der Plugin.xml-Datei erfolgt nach dem Lazy-Initialized Prinzip. Es werden erst die deklarierten Erweiterungen instantiiert wenn sie benötigt werden. Dies hat den Vorteil, dass zur Laufzeit nur sehr wenig Speicherressourcen benötigt werden.

2.3.3 Model-View-Control Pattern

Das Model-View-Control (MVC) Pattern wird durch drei Komponenten umgesetzt. Die Model-Komponente stellt das Anwendungsobjekt dar, das den Zustand der Anwendung speichert. Die View-Komponente stellt die Daten auf dem Bildschirm dar. Bei Änderungen der Daten signalisiert die Model-Komponente diese den abhängigen Views. Daraufhin erhält die View-Komponente die Möglichkeit, selbstverantwortlich den Zustand des Anwendungsobjekt wiederzugeben. Mit diesem Ansatz wird es ermöglicht, mehrere Views an ein Model zu binden, um verschiedene Präsentationen anzubieten. Die Controller-Komponente legt fest, wie auf Benutzereingaben in der View-Komponente reagiert werden soll. Will man die Antwortstrategie verändern, so kann zur Laufzeit die Controller-Implementierung gewechselt werden. Dadurch kann ein View beispielsweise auch abgeschaltet werden, indem man einen Controller zuweist, der alle Benutzereingaben ignoriert (vgl. Gamma u. a. (2011)).

3 Konzept und Architektur

In diesem Kapitel werden die Anforderungen an die Entwicklungsumgebung aufgezeigt (Abschnitt 3.1). Anschließend wird Abgegrenzt in welchem Kontext die Software entwickelt wird (Abschnitt 3.2). Zuletzt wird die Architektur beschrieben, die sich aus den Anforderungen und dem Kontext ergibt (Abschnitt 3.3).

3.1 Anforderungen

3.1.1 Modellbildungsprozess

Vielseitigkeit bei der Modellierung Um Proben eines Experiments auf ein Modell abzubilden sind oftmals viele Komponenten notwendig, da ein zu untersuchendes System oft aus vielen Atomen zusammengesetzt ist. Damit eine übersichtliche Modellierung erfolgen kann, muss ein System in logisch abgegrenzte Gruppen unterteilt werden. Die Verwaltung der einzelnen Komponenten muss daher so konzipiert werden, dass eine übersichtliche Handhabung aller modellierten Atome bestehen bleibt.

Positionierung der Atome Bei der Anordnung der einzelnen Atome in einem System ist eine möglichst genaue Positionierung aller zu modellierenden Atome erforderlich. Kleinste Abweichungen bei der Berechnung der Positionen können zu erheblichen Abweichungen der Ausgangsgrößen führen. Diese Fehler sind oftmals bei der Analyse der Simulationsergebnisse nur schwer ausfindig zu machen. Um diese Fehler bereits bei der Modellierungsphase zu reduzieren, sollen alle Positionierungen durch Vektorrechnungen unterstützt werden.

Parametrisierung der Komponenten Je nach Element haben die einzelnen Atome unterschiedlich zu parametrisierende Wechselwirkungen (siehe Abschnitt 2.2.3). Um nicht jedes Atom und dessen Wechselwirkungen einzeln zu parametrieren, soll ein Atom oder eine Gruppe von Atomen den entsprechenden Parametern zugeordnet werden. Dadurch können die Parameter für die Wechselwirkungen separat definiert werden und ermöglichen eine spätere Änderung an zentraler Stelle.

3.1.2 Simulation

Aktionen der Simulationsausführung Da die Techniken eines RTMs stets weiterentwickelt werden, soll die Simulation so gestaltet werden, dass möglichst vielseitige Experimente nachgebildet werden können. Um dies zu erreichen wird der Prozess eines Experiments in unterschiedliche Aktionen untergliedert. Abstrakt kann ein Experiment in drei Aktionsklassen unterteilt werden, die für unterschiedliche Anforderungen ausgetauscht werden können.

Austausch von Wechselwirkungen: Um das Bewegungsverhalten einzelner Atome zu untersuchen, können Wechselwirkungen definiert werden, die das Bindungsverhalten zu Nachbaratomen nachbilden. So können Positionsänderungen berechnet werden, die je nach Zustand des Systems energetisch günstiger sind. Dabei gibt es verschiedene Wechselwirkungen, die unterschiedliche Vor- und Nachteile mit sich bringen. Je nach Anforderungen an die Simulation sollen diese ausgetauscht werden können.

Austausch von Spitzenbewegungen: Da ein RTM vielseitige Untersuchungen einer Probe zulässt, soll auch die Simulation anpassbar auf verschiedene Bewegungsabläufe sein. Üblicherweise werden konstante Bewegungen modelliert, die das Verhalten der Spitze nachbilden. Dabei wird die Spitze in konstanter Richtung und Geschwindigkeit über die Probe gesteuert, um ein Messsignal aufzuzeichnen. Allerdings kann die Spitze auch abhängig des gemessenen Signals in Z-Richtung modifiziert werden, um ein konstant-Strom-Modus (Siehe Kap. 2.2.1) zu realisieren. Um dies zu erreichen, muss die Art der Bewegung eines oder mehrerer Atome austauschbar sein.

Austausch von Messmethoden: Bei einem simulierten Prozess können je nach Fokus des Experiments unterschiedliche Ausgangsgrößen von Interesse sein. Üblicherweise stellt der Tunnelstrom zwischen Spitze und Probe die zu untersuchende Größe dar, um ein Experiment aus dem RTM nachzubilden und mit bekannten Messwerten zu vergleichen. Allerdings können auch andere Größen, wie z.B. die Energie einzelner Atome bei unterschiedlichen Zuständen von Interesse sein. Daher muss die Wahl der Messmethode einer Simulation entsprechend dem zu simulierenden Experiment anpassbar sein.

Ausführungsmethoden Die Simulation soll so gestaltet werden, dass eine parallele Berechnung durchgeführt werden kann, um einen schnelleren Simulationsdurchlauf zu erzielen.

3.1.3 Entwicklungsumgebung

Gestaltung und Bedienbarkeit Die komplette grafische Oberfläche soll einem einheitlichen und übersichtlichen Bedienkonzept folgen. Dadurch soll eine intuitive Bedienung und eine schnelle Einarbeitung ermöglicht werden.

Einheitliche Schnittstellen der Komponenten Die einzelnen Komponenten der Entwicklungsumgebung sollen so realisiert werden, dass sie möglichst durch einheitliche Schnittstellen verwendet werden können.

Erweiterbar für verschiedene Anforderungen Die Komponenten sollen so realisiert werden, dass eine Erweiterung der Entwicklungsumgebung oder einzelner Komponenten möglich ist, ohne bereits getestete Komponenten verändern zu müssen.

Unterstützung bei Simulationsstudien Die Entwicklungsumgebung soll so gestaltet werden, dass möglichst viele Teile einer Simulationsstudie von der Umgebung unterstützt und vereinfacht werden.

3.2 Abgrenzung

3.2.1 Mögliche Einsatzszenarien

Die in dieser Arbeit zu implementierende Entwicklungsumgebung soll die Durchführung von Simulationsstudien im Bereich der Nano-Struktur-Forschung unterstützen und vereinfachen. Die Simulation kann bei ersten Machbarkeitsstudien verwendet werden, um Aussagen über bisher unbekannte Systeme zu treffen und so bei der Entscheidung helfen, das unbekannte System mit aufwendigen Experimenten zu untersuchen.

Ein weiteres Einsatzszenario besteht in der Untersuchung experimenteller Systeme, die bereits untersucht wurden. Durch die Entwicklung von Simulationsstudien kann versucht werden, eine ausreichend gute Beschreibung des Experiments in ein Simulationsmodell zu überführen. Bei Übereinstimmungen der Ergebnisse von Simulation und Experiment können bessere Aussagen über das reale System getroffen werden, sowie Bewegungsabläufe einzelner Atome präzise visualisiert werden.

3.2.2 Arbeitsablauf einer Simulationsstudie

Die Basis für den Einsatz der Entwicklungsumgebung ist ein zuvor abgegrenztes System. Dabei ist vorab zu klären, welcher Bereich des Systems modelliert werden soll und welche Temperatur den Vorgang beeinflusst (siehe Abschnitt 2.1.2: Problemdefinition). An dieser Stelle setzt die Entwicklungsumgebung zur Durchführung der Simulationsstudie an, um das System in ein geeignetes Modell zu überführen.

In der ersten Phase wird die Anordnung der Atome des Experiments modelliert und in die verschiedenen Komponenten (z.B. Spitze, freie Atome, Substrat-Lagen, usw.) zusammengefasst

(siehe Abschnitt 2.1.2: Formulierung eines Modells). Die Modellierung kann an dieser Stelle durch grafische Visualisierung überprüft und abgeändert werden, um fehlerhafte Übertragungen von Realität in das Modell zu reduzieren (siehe Abschnitt 2.1.2: Problemdefinition).

In der zweiten Phase wird das Prozessverhalten definiert. An dieser Stelle werden Interaktionen und deren zeitliche Abfolge definiert, die sich innerhalb des Modells abspielen. Darunter fallen:

- die Wechselwirkungen der Atome zueinander, die eine variable Positionsveränderung mit sich führen (wie beispielsweise das Adatom),
- eine fixe Positionsveränderung in fest definierter Richtung zur Modellierung von Spitzenverhalten,
- das Auslesen von Parametern einzelner Atome zur Aufnahme von Messdaten.

Die Parameter der einzelnen Wechselwirkungen können auch hier graphisch untersucht werden, um vorab eine Einschätzung über die Eingangsgrößen der Simulation zu erhalten.

In der nächsten Phase wird das Verhalten der Simulationsausführung definiert. Dabei kann angegeben werden, wie die Prozessdurchführung parallelisiert wird und in welcher Art sie verteilt ausgeführt werden soll.

Anschließend wird die Simulation durchgeführt (siehe Abschnitt 2.1.2: Entwurf und Durchführung der Simulationsversuche). Auch an dieser Stelle kann die Simulationsdurchführung graphisch visualisiert werden, um somit vorzeitige Fehler zu erkennen oder das Verhalten der Atome zu analysieren.

Nach Abschluss der Simulationsausführung wird ein detaillierter Simulationsbericht erstellt, der alle Parameter des Modells und der Simulation beinhaltet (siehe Abschnitt 2.1.2: Dokumentation und Präsentation der Ergebnisse).

In der letzten Phase können die produzierten Messergebnisse analysiert werden (siehe Abschnitt 2.1.2: Analyse der Simulationsergebnisse). Diese werden zum einen als Kontrastbild dargestellt, in dem die X- und Y-Position der Spitze aufgetragen wird und über den Kontrast die Messgröße abbildet. Zum andern können die Messwerte jeweils linienweise in einem X-Z-Diagramm dargestellt werden, um eine genauere Analyse der Messwerte zu unterstützen. Die beiden Arten der Darstellung erlauben nur eine schnelle Analyse der Ergebnisse. Um eine genauere Untersuchung zu unterstützen, wird eine Export-Methode angeboten. Dadurch können die Daten mit den gleichen Werkzeugen[1] untersucht werden, wie die Ausgangssignale eines RTMs. Dies ermöglicht eine genaue Gegenüberstellung von Ergebnissen des Experiments und der Simulation.

[1] Softwaretools zur Analyse der physikalischen Ausgangssignale der RTMs

3.2.3 Verwendete Techniken

Relaxation von Atomen Um die Dynamik eines Systems mit ortsveränderlichen Atomen zu simulieren wurde eine Relaxation[2] verwendet. Diese wird mit dem Metropolis Algorithmus umgesetzt. Der Algorithmus stellt eine Monte-Carlo-Methode zur Erzeugung von Markov-Ketten[3] entsprechend der Boltzmann-Verteilung[4] dar. Dadurch ergibt sich für die Atome ein wahrscheinlichkeitsgewichteter Pfad durch den Zustandsraum. Der Algorithmus ergibt sich für ein Atom wie folgt:

- $\vec{x}' \in \mathbb{R}^3$ die Position des Atoms. Δx die maximale Schrittweite.
- $\vec{g}_{[-1,1]} \in \mathbb{R}^3$ ein gaußverteilter Zufallsvektor mit Komponenten -1,1 und $r_{[0,1]} \in [0,1]$ eine gleichmäßig verteilte Zufallszahl.
- $E(\vec{x})$ die Energie eines Atoms an einer Position $\vec{x}$ und $p(\vec{x}, \vec{x'})$ eine temperaturabhängige Übergangswahrscheinlichkeit eines Atoms zwischen zwei Orten $\vec{x}$ und $\vec{x'}$.

1. Auswahl einer neuen Position $\vec{x'} = \vec{x} + \Delta x \cdot \vec{g}_{[-1,1]}$
2. Berechnen der Energie $E'(\vec{x'})$
3. Berechnen der Übergangswahrscheinlichkeit
$$p(\vec{x}, \vec{x'}) = \begin{cases} 1 & \text{für } E' < E \\ exp\{-(E - E'/kT)\} & \text{für } E' > E \end{cases}$$
4. Die Position $\vec{x'}$ wird als neue Position übernommen, wenn $r_{[0,1]} < p(\vec{x}, \vec{x}')$

Für jedes Atom wird $\vec{g}_{[-1,1]}$ und $r_{[0,1]}$ neu bestimmt. Eine Position mit höherer Energie wird durch die Boltzmann-Verteilung bestimmt, wobei T die Temperatur des Systems und k die Bolzmann-Konstante ist (vgl. Wolter (2009)). Die Energie der Atome wird durch die Wechselwirkungen (siehe Abschnitt 2.2.3) der chemischen und magnetischen Bindung berechnet.

[2] Übergang eines Systems in einen Gleichgewichtszustand
[3] Bilden eine statistische Aussage über den Folgezustand anhängig vom gegenwärtigen Zustand
[4] Verteilung der Energie in Abhängigkeit zur Temperatur

Berechnung des Tunnelstroms Um die Ergebnisse der Simulation mit denen des Experiments vergleichen zu können, wurde das Tersoff-Hamann-Model zur Berechnung des Tunnelstroms verwendet. Der Ansatz setzt voraus, dass die magnetische Ausrichtung der Spitzen und Probenatome bekannt sind. Der Tunnelstrom I_{TS} zwischen dem Spitzenatom TP und den Probenatomen i ist abhängig von den Atompositionen r_{TP} und r_i, den Magnetisierungen P_{TP} und P_i, sowie den magnetischen Ausrichtungen S_{TP} und S_i.

$$I_{TS}(r_{TP}) \propto \sum_i (1 + P_{TP} P_i S_{TP} \cdot S_i) \exp^{-2\kappa |r_{TP} - r_i|}$$

Für die Tunnelstrom-Berechnung mit einem Adatom wurde die Formel zwischen Spitze und Probe, Spitze und Adatom sowie Adatom und Probe angewendet (I_{TAS}). Der Tunnelstrom ergibt sich dadurch wie folgt:

$$I(r_{TP}, r_A) = I_{TS}(r_{TP}) + I_{TAS}(r_{TP}, r_A)$$

Um den Aufwand zu reduzieren, wurden nur Atome innerhalb eines *cutoff* Radius[5] berechnet (vgl. Wolter (2014)). Der Algorithmus wurde übernommen und für die Anwendung angepasst. Die wesentlichen Berechnungen wurden nicht geändert.

3.3 Architektur der Komponenten

Die Entwicklungsumgebung wird in mehrere Komponenten unterteilt. So wird das Gesamtproblem in unabhängige Teilprobleme zerlegt. Dadurch wird gleichzeitig bessere Austauschbarkeit und Erweiterbarkeit erreicht. Die Komponenten werden nach dem MVC-Pattern (siehe Abschnitt 2.3.3) entworfen. Alle Komponenten, die eine Datenhaltung beinhalten, werden in zwei Teilkomponenten zerlegt (siehe Abbildung 3.1). Die eine Teilkomponente enthält die Datenhaltung sowie Funktionen zur Datenverarbeitung (blau dargestellt). Die andere Teilkomponente präsentiert die Daten und stellt die Ereignisverarbeitung bei Benutzereingaben bereit (orange dargestellt).

3.3.1 Modellierung des Systems

Die Environment-Komponente realisiert die Datenhaltung aller Objekte, die für die Modellierung des Systems benötigt werden. Der Aufbau untergliedert sich in drei Teilkomponenten: die Objekt-Verwaltung, Objekt-Eigenschaften und Vektor-Verwaltung (siehe Abbildung 3.2).

[5] Definiert maximalen Abstand der nächsten Nachbaratome für eine Berechnung

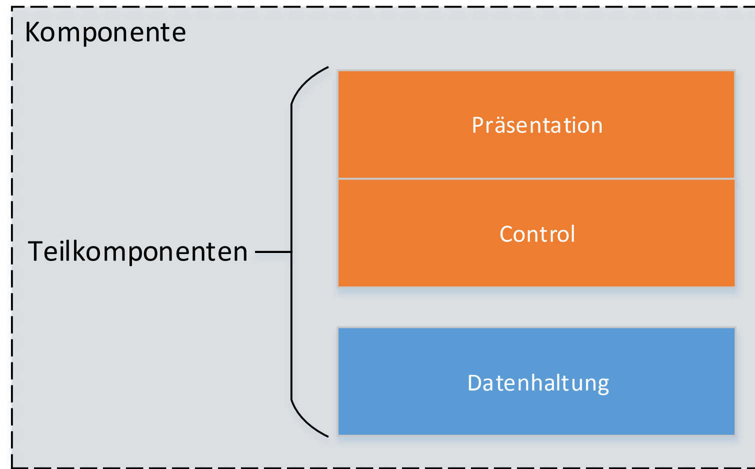

Abbildung 3.1: Aufbau einer Komponente der Entwicklungsumgebung

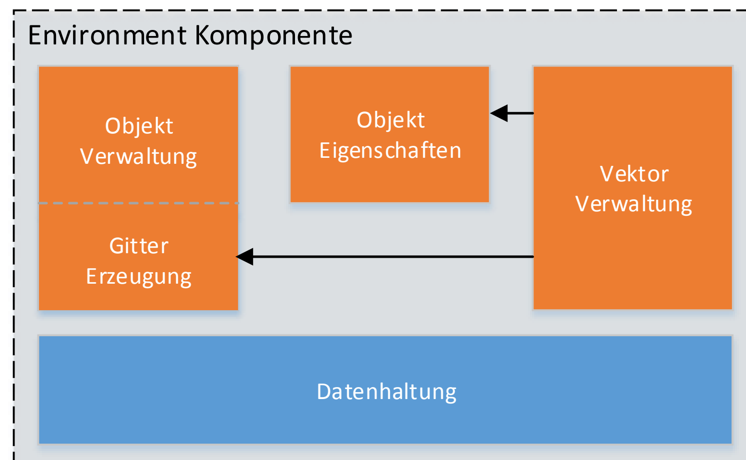

Abbildung 3.2: Aufbau der Environment-Komponente

Vektor-Verwaltung Um ein System im dreidimensionalen Raum zu modellieren, ist es oft hilfreich Vektoren zu verwenden, um Position oder Ausrichtung von Objekten zu definieren. Die Vektor-Verwaltungskomponente stellt Funktionen zur Verfügung, mit denen Vektoren erstellt werden können. Darüber hinaus ist es möglich, erstellte Vektoren mit Operationen zu manipulieren. In dieser Teilkomponente werden folgende Funktionen realisiert:

Der zu erstellende Vektor $\vec{a}$ wird komponentenweise mit einem zuvor definierten Vektor $\vec{b}$ addiert:

$$\vec{a} + \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{pmatrix}$$

Der zu erstellende Vektor $\vec{a}$ wird komponentenweise mit einem zuvor definierten Vektor $\vec{b}$ subtrahiert:

$$\vec{a} - \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} - \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 - b_1 \\ a_2 - b_2 \\ a_3 - b_3 \end{pmatrix}$$

Der zu erstellende Vektor $\vec{a}$ wird komponentenweise mit einem Skalar r multipliziert:

$$r \cdot \vec{a} = r \cdot \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} ra_1 \\ ra_2 \\ ra_3 \end{pmatrix}$$

Der zu erstellende Vektor $\vec{a}$ wird komponentenweise mit einem Skalar r dividiert:

$$\vec{a}/\, r = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} /\, r = \begin{pmatrix} a_1/\, r \\ a_2/\, r \\ a_3/\, r \end{pmatrix}$$

Der zu erstellende Vektor $\vec{a}$ wird um die x-Achse mit einem Winkel α rotiert:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

Der zu erstellende Vektor $\vec{a}$ wird um die y-Achse mit einem Winkel α rotiert:

$$R_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix}$$

Der zu erstellende Vektor $\vec{a}$ wird um die z-Achse mit einem Winkel α rotiert:

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Objekt-Verwaltung Nachdem das Problem abgegrenzt ist und alle erforderlichen Systemparameter und Daten ermittelt wurden, kann mit dieser Teilkomponente das System modelliert werden. Zu Beginn der Modellierung werden alle Objekte (Atome) des Systems, die miteinander in Wechselwirkung stehen sollen, definiert.

Um komplexe experimentelle Systeme zu modellieren ist es hilfreich nicht alle Objekte auf einer Hierarchieebene (Auflistung aller Objekte) zu organisieren. Daher wird eine Baumstruktur verwendet, die es ermöglicht ein System in gedankliche Abschnitte zu unterteilen. Das Wurzelelement der Baumstruktur repräsentiert das zu simulierende Gesamtsystem. Als Äste werden Gruppen hinzugefügt, die mit Atomen als Blätter gefüllt werden. So kann eine Probe mit mehreren atomaren Lagen in Gruppen unterteilt und damit übersichtlich verwaltet werden.

Damit Gitterstrukturen der Proben schneller und fehlerfrei erzeugt werden können, ist es möglich Basisvektoren zu definieren, mit denen Gitter in beliebiger Größe algorithmisch erstellt werden. Dazu wird das Konzept des Bravais-Gitters verwendet, welches das Raumgitter spezifiziert, auf dem die Atome des Kristalls angeordnet sind. Ein dreidimensionales Bravais-Gitter besteht aus allen Punkten: $R = n_1 a_1 + n_2 a_2 + n_3 a_3$ mit ganzen Zahlen n_i und linear unabhängigen Vektoren $a_i, i = 1, 2, 3$ (vgl. Gross und Marx (2012)). Damit lassen sich im dreidimensionalen Raum 16 unterschiedliche Bravais-Gitter erzeugen, welche alle gängigen Strukturen (siehe Abschnitt 2.2.2) beinhaltet. Da sich Proben oftmals aus mehreren Lagen eines Gittertyps zusammensetzen lassen wird der Algorithmus so implementiert, dass er für zwei- sowie dreidimensionale Strukturen verwendet werden kann.

Objekte-Eigenschaften Für die Parametrisierung einzelner oder mehrerer Atome bietet diese Komponente die Möglichkeit, Eigenschaften zur Laufzeit zu verändern oder zu ergänzen. Um den experimentellen Aufbau nachzubilden, werden alle Atome mit einem Ortsvektor eines kartesisches Koordinatensystem positioniert. Da das Modellieren eines Systems mit

vielen Objekten durch manuelle Positionierung sehr langwierig sein kann, bietet die Objekt-Verwaltung mit der Gitter-Erzeugung die Möglichkeit, alle dort erstellten Objekte automatisch zu positionieren. Damit einzelnen Atomen element-spezifische Wechselwirkungen zugewiesen werden können, müssen diese mit einem eindeutigen Element-Typ klassifiziert werden. So lassen sich die Parameter der Wechselwirkungen den bestimmten Atomen zuordnen. Durch die hierarchische Anordnung der Atome durch die Baumstruktur ist es ebenfalls möglich, Eigenschaften wie Element-Typ oder Position für alle Atome einer Gruppe zu definieren. So können beispielsweise Atome einer Gruppe um einen definierten Vektor verschoben werden oder die magnetische Ausrichtung definiert werden.

3.3.2 Prozessabbildung

Die Simulationskomponente realisiert die Datenhaltung aller Aktionen und deren zeitliche Abfolge, die den Simulationsprozess abbilden. Der Aufbau untergliedert sich in drei Teilkomponenten: die Simulationsverwaltung, Parameter-Verwaltung und Vektor-Verwaltung (siehe Abbildung 3.3).

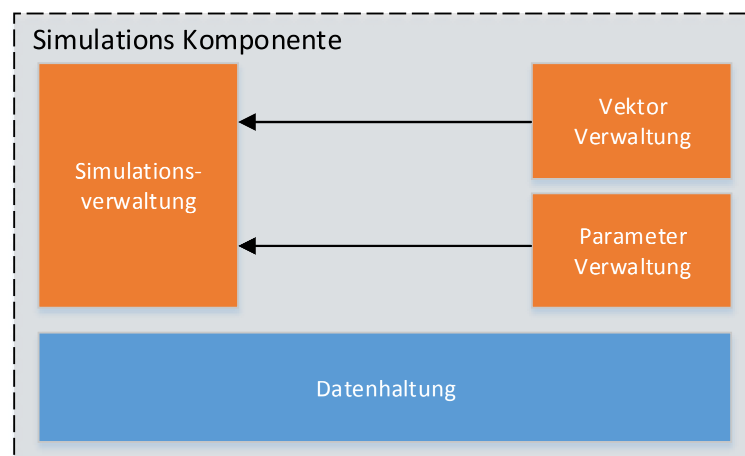

Abbildung 3.3: Aufbau der Simulationskomponete

Parameter-Verwaltung Für die verschiedenen Wechselwirkungen (siehe Abschnitt 2.2.3), die zwischen den einzelnen Atomen bestehen und deren Bindungsenergie beschreiben, werden unterschiedlichste Parameter benötigt. Damit weitere Wechselwirkungen implementiert werden können, die im aktuellen Entwicklungsstadium noch nicht realisiert sind, wird das Parameterhandling generisch gestaltet. So kann für einen element-spezifischen Bindungstyp

(z.B. Eisen-Atom) ein ParameterSet angelegt werden, das unterschiedliche Wechselwirkungen als Kategorien (chemische- und magnetische Bindung) beinhaltet. Innerhalb einer Wechselwirkungskategorie werden anschließend die spezifischen Parameter für die Wechselwirkungen definiert.

Vektoren-Verwalten Wie bereits bei der Modellierung des Systems beschrieben (siehe Abschnitt 3.3.1), kommen auch in der Prozessmodellierung Vektoren zum Einsatz. An dieser Stelle werden sie zum Abbilden von Positionsänderungen verwendet.

Simulationsverwaltung Die Prozessabbildung erfolgt mithilfe eines Simulationsobjektes (siehe Abbildung 3.4). Dort wird zunächst die Anzahl der zu durchlaufenden Simulationsschritte definiert, sowie das Environment angegeben, das den atomaren Aufbau der zu untersuchenden Probe repräsentiert. Anschließend werden Aktionen dem Simulationsobjekt hinzugefügt, die das dynamische Verhalten des Prozesses nachbilden. Jede hinzugefügte Aktion benötigt zum einen ein Schritte-Intervall, welches angibt, zu welchen Zeitpunkten der Simulationsausführung die Aktion jeweils ausgeführt werden soll. Zum anderen wird eine Priorität angegeben, die eine konkrete Ausführungsreihenfolge realisiert, wenn zwei Aktionen zum gleichen Zeitpunkt ausgeführt werden sollen. Sollte zudem vom Benutzer eine gleiche Priorität bei überschneidenden Aktionsausführungen definiert werden, so wird die Aktion ausgeführt, die zuerst hinzugefügt wurde. Als nächstes wird eine konkrete Realisierung einer Aktionsklasse ausgewählt. Im derzeitigen Entwicklungsstand wurde für jede der drei Aktionsklassen (siehe Abschnitt 3.1.2) eine konkrete Realisierung implementiert.

Die Energieberechnung der **Relaxations-Aktion** wurde mit dem Morse-Potential und einem magnetischen Austausch berechnet (siehe Abschnitt 3.2.3). Um die Relaxation in das Simulationsobjekt einzubinden, werden vorab die Parameterdefinitionen für diese Wechselwirkungen benötigt. Diese können über die ParameterSets definiert werden. Für das Morse-Potential wird beispielsweise eine Kategorie *Morse* mit den Parametern U_0, R_0 und *Alpha* benötigt. Für die magnetische Wechselwirkung wird in dem ParameterSets eine Kategorie *Exchange* mit den Parametern J_0, R_0 und *Alpha* benötigt. Für jeden Element-Typ (zum Beispiel Chrom, Kupfer, Eisen usw.) wird jeweils ein ParameterSet angelegt, das die spezifische Bindung charakterisiert. Nachdem die benötigten ParameterSets für die Wechselwirkungen angelegt wurden, kann die Relaxation erstellt werden. Aus dem zugeordneten Environment werden zwei Mengen von Atomen ausgewählt (siehe Abbildung 3.5): die Menge der zu relaxierenden Atome und die Menge der Atome, die für die Energieberechnung der zu relaxierenden Atome abhängig sind. Bei der Manipulation eines Atoms entlang einer Oberfläche mithilfe der

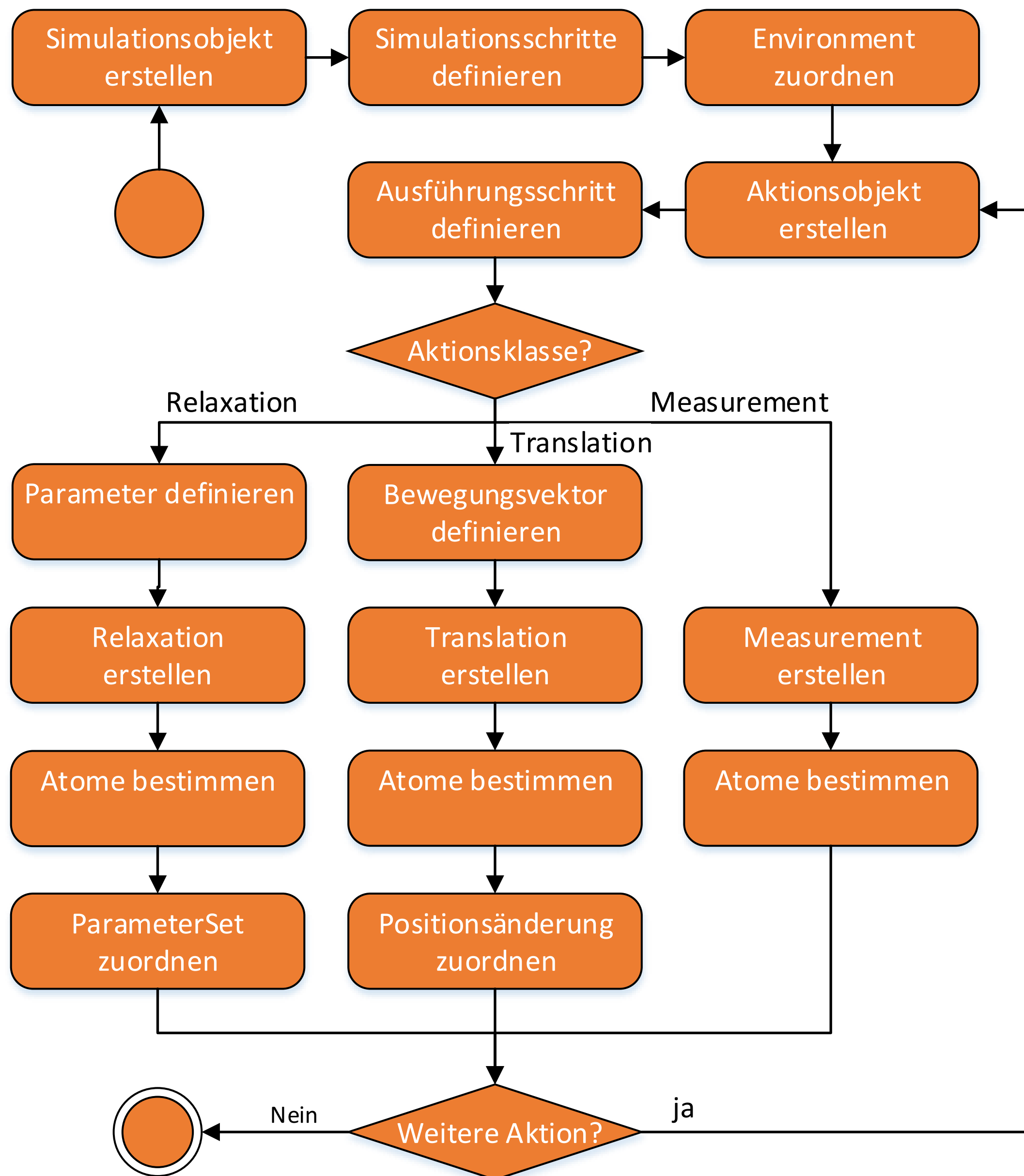

Abbildung 3.4: Ablauf der Prozessmodellierung

Spitzenbewegung besteht die Menge der zu relaxierenden Atome aus dem Adatom (M_{Relax}). Die Menge der abhängigen Atome besteht aus den Atomen der Spitze und den Atomen der Oberfläche $M_{Dependent}$, über die das Atom manipuliert werden soll.

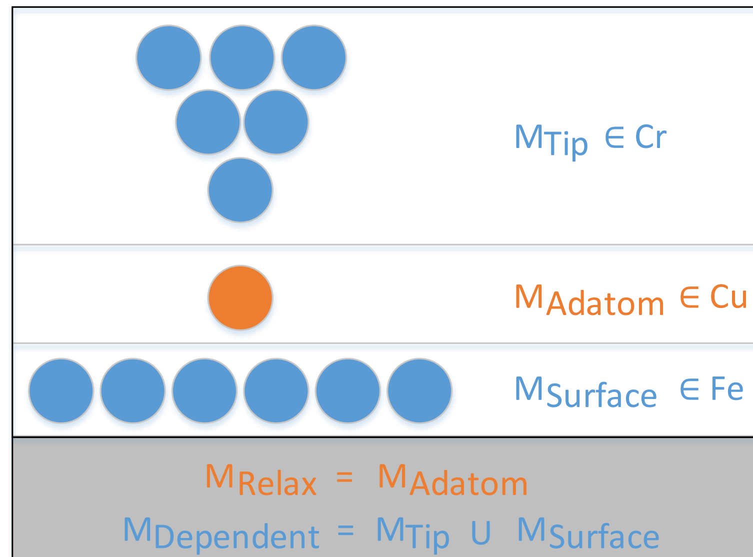

Abbildung 3.5: Atommengen eines Systems

Die konkrete Implementierung der **Translations-Aktion** realisiert eine Bewegung eines oder mehrerer Atome in vordefinierter Richtung und Weite. Die Bewegung wird bei jeder Aktionsausführung fortgesetzt. Um die Translations-Aktion in das Simulationsobjekt einzubinden, muss vorab ein Vektor definiert werden, mit dem die Bewegung modelliert wird. Die Vektorrichtung entspricht der Bewegungsrichtung. Die Vektorlänge gibt die Bewegungsweite einer Aktionsausführung an. Nachdem der benötigte Vektor angelegt wurde, kann die Translation erstellt werden. Aus dem zugeordneten Environment wird eine Menge von Atomen ausgewählt, der die Bewegung zugeordnet werden soll. Bei der Manipulation eines Adatom mithilfe der Spitzenbewegung besteht die Menge der zu bewegenden Atome aus den Atomen der Spitze (M_{Tip}). Der Bewegungsablauf entspricht der Spitzenbewegung entlang einer Scan-Linie mit dem konstant-Höhe-Modus (siehe Abschnitt 2.2.1).

Die Implementierung der **Measurement-Aktion** realisiert das Aufzeichnen des Tunnelstroms (siehe Abschnitt 3.2.3). Aus dem zugeordneten Environment werden die Atome angegeben, die für die Berechnung des Tunnelstroms benötigt werden. Dafür wird die Menge Anodenatome, die Menge der Kathodenatome sowie die Menge der Atome zwischen den Elektroden, die die Stromübertragung beeinflussen. Für die oben beschriebene Adatom Mani-

pulation besteht die Kathode aus den Spitzenatomen (in Abbildung 3.5 M_{Tip}), die Anode aus den Atomen der Probenlagen ($M_{Surface}$) und die zwischen liegenden Atome sind in diesem Fall nur das Adatom (M_{Adatom}).

3.3.3 Simulationsausführung

Die Distributor-Komponente realisiert die Verteilung und Ausführung der Simulation. Sie besteht aus drei Teilkomponenten (siehe Abbildung 3.6).

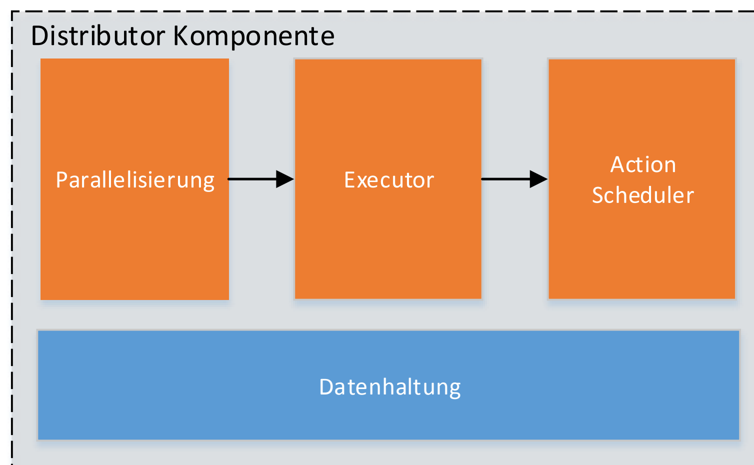

Abbildung 3.6: Aufbau der Distributor-Komponente

Parallelisierung der Simulationsausführung Bei der parallelen Ausführung von Simulationen werden Prozesse simuliert, die jeweils unterschiedliche Anfangskonfigurationen besitzen. Das bedeutet im Bezug auf die Manipulation eines Adatoms in paralleler Simulationsausführung, dass jeweils mit einer unterschiedliche Konfiguration der Startposition des Adatoms und der Spitze ausgeführt wird. So können in verschiedenen Simulationen unterschiedliche Linien-Scans simuliert werden. Dafür bietet die Distributor-Komponente die Möglichkeit für jede Simulation die Parameter sowie die Parameter des Modells zu ändern.

Bei der Erzeugung eines Distributor-Objektes wird vorerst angegeben, wie viele Simulationsinstanzen benötigt werden. Anschließend muss dem Distributor eine initiale Simulationsinstanz zugeordnet werden, welche entsprechend oft vervielfältigt wird. Damit die Simulationen sich voneinander unterscheiden und nicht die gleichen Ergebnisse liefern, werden Aktionen hinzugefügt, die für eine Unterscheidung der Simulationsprozesse sorgen. Die Distributor-Aktionen haben dabei Zugriff auf das Environment, sowie die Aktionen der Simulationen. So kann

für die Simulationsinstanzen eine Änderung der Atomanordnung sowie eine Änderung des Prozesses (dynamische Verhalten des Systems) erfolgen. Die Distributor-Aktionen werden bei Simulationsbeginn einmal Aufgerufen und haben während der Simulationsdurchführung keinen Einfluss mehr auf die Simulationsinstanzen und dessen Environments.

Im aktuellen Entwicklungsstand wurde eine Aktion implementiert, die einen parallelen Linien-Scan realisiert. Dafür wird zum einen eine Menge von Atomen der initialen Simulation ausgewählt. Zum anderen wird ein Vektor $\vec{a}$ angegeben, der eine Verschiebung der Atome in den einzelnen Simulationen erzeugt. Bei Ausführung des Distributors wird für jede neue Simulationsinstanz mit dem Index i die Menge von Atomen um $\vec{a} \cdot i$ verschoben. So unterscheiden sich alle Environments der neuen Simulationsinstanzen um die vorgegebene Verschiebung. Bei der Manipulation eines Adatoms besteht die Menge der Atome aus dem Adatom und den Spitzenatomen. So wird für jede Scan-Linie eine unterschiedliche Anfangskonfiguration des Environments erzeugt.

Verteilung der Simulation Die Verteilungskomponente wurde so konzipiert, dass eine lokale sowie entfernte Ausführung der Simulationen realisiert werden kann. Das Distributor-Objekt reicht die Simulationsinstanzen an einen Executor weiter, der für die Verteilung verantwortlich ist. Der implementierte Executor verwendet einen Threadpool, der die einzelnen Simulationen jeweils auf die lokal zur Verfügung stehenden Prozessorkerne verteilt. Für die Ausführung in verteilten Systemen kann jedoch ein weiterer Executor implementiert werden, der die Datenübertragung und Koordinierung realisiert.

Ausführung der Simulation Die Ausführung erfolgt nach dem Modell der zeitgesteuerten Simulation (siehe Abschnitt 2.1.4). Um die Ausführung der einzelnen Simulations-Aktionen vorab einzustellen wird ein Intervall definiert, dass angibt nach wie vielen Simulationsschritten die jeweilige Aktion ausgeführt werden soll. Um auch bei Schrittgleichen Ausführungen mehrerer Aktionen eine wohldefinierte Folge zu erreichen, werden zusätzlich Prioritäten definiert. Beim Start einer Simulation werden alle dort in einer Liste hinterlegten Aktionen zu einem Scheduler übertragen, der die Ausführung übernimmt. Der Scheduler stellt die Simulationsuhr dar (siehe Abschnitt 2.1.3). Dieser organisiert die Aktionen so, dass ein schneller Zugriff bei der Ausführung gegeben ist.

3.3.4 Analyse des Systems und Simulationsüberwachung

Die Monitor-Komponente unterteilt sich in drei Teilkomponenten, die für die Visualisierung zuständig sind (siehe Abbildung 3.7).

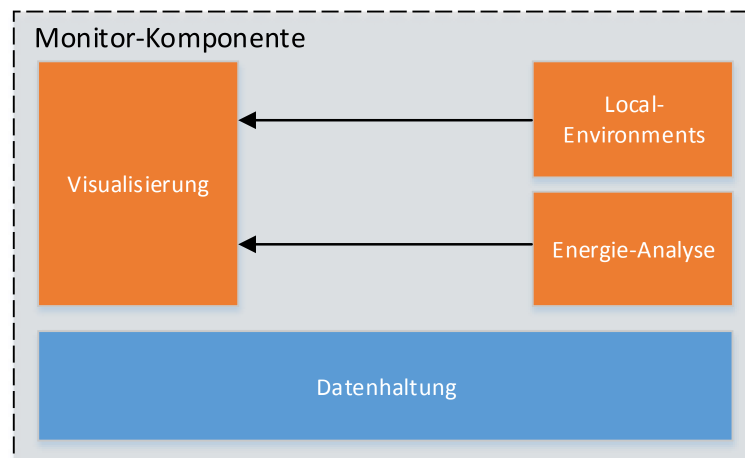

Abbildung 3.7: Aufbau der Monitor-Komponente

Visualisierung Die Visualisierungskomponente erlaubt eine dreidimensionale Darstellung des Environments, womit bereits in der Modellierungsphase Fehler vermieden werden können. Änderungen am Environment werden direkt in die Darstellung übernommen. Dabei werden alle Objekte des Environments anhand ihrer Positionseigenschaften in einem dreidimensionalen kartesischen Koordinatensystem dargestellt.

Energie-Analyse Zur Einstellung der zu verwendenden Wechselwirkungen können die Energieniveaus des Systems visualisiert werden. Dazu kann die Energie der gesamten Probe als Landschaftsbild dreidimensional visualisiert oder Ausschnitte davon in einem Liniendiagramm detailliert untersucht werden.

Local-Environments Diese Komponente stellt eine auswählbare Auflistung aller modellierten Environments bereit. Das jeweils ausgewählte Environment wird in der Visualisierungskomponente angezeigt.

3.3.5 Dokumentation der Ergebnisse und Auswertung

Die Analysis-Komponente gliedert sich in drei Teilkomponenten. Nach der Durchführung einer Simulation generiert die Analysis-Komponente einen umfangreichen Report. Für die Evaluation der Ergebnisse stehen Werkzeuge zur Visualisierung bereit. Um die Ergebnisse detaillierter zu untersuchen, können die gewonnenen Daten exportiert werden um sie mit anderen Programmen weiter zu bearbeiten (siehe Abbildung 3.8).

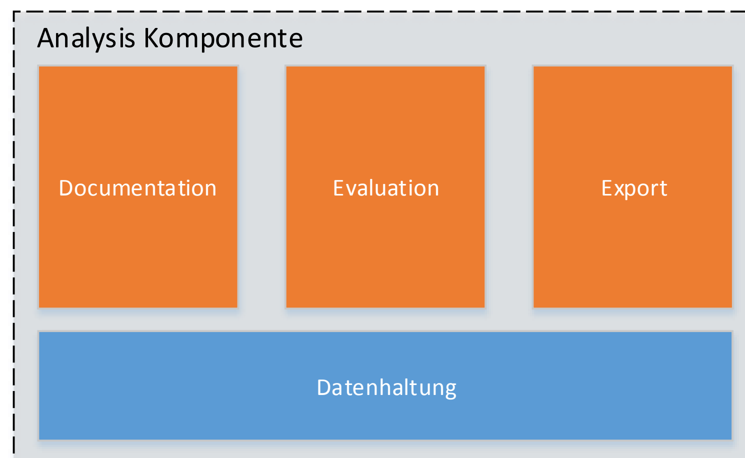

Abbildung 3.8: Aufbau der Analysis-Komponente

Dokumentation der Ergebnisse Nach Simulationsdurchführung erzeugt die Analysis-Komponente einen HTML-Report, der alle relevanten Einstellungen der Simulation und des Distributors beinhaltet. So können im Nachhinein Simulationen reproduziert werden.

Auswertung der Ergebnisse Eine Simulationsausführung kann aus vielen einzelnen Simulationen bestehen, die jeweils Artefakte zur Speicherung der Daten erzeugen. Die Analysis Komponente bietet die Möglichkeit einzelne Artefakte in einem zweidimensionalen Chart darzustellen, um so von jeder Simulation die Ergebnisse gesondert betrachten zu können.

Eine weitere Möglichkeit die Simulationsergebnisse zu visualisieren besteht darin, alle vorhanden Artefakte zu einem Gesamtbild zusammenzufügen. Dabei werden die vorhandenen Daten nach der Messposition in X-, Y-Richtung aufgetragen. Die einzelnen Messwerte werden als Graustufen angezeigt.

Konvertierung der Ergebnisse Um weitere Analysemöglichkeiten zu gewährleisten, können die Daten des Gesamtbildes in ein Datenformat exportiert werden, das von vielen Tools, zur Analyse von Rastertunnel-Aufnahmen, eingelesen werden kann.

4 Realisierung

In diesem Kapitel wird zuerst ein kurzer Überblick über die verwendeten Bibliotheken gegeben (siehe Abschnitt 4.1). Anschließend wird beschrieben mit welchen Techniken die Aufteilung der Komponenten realisiert wurde um die Austausch- und Erweiterbarkeit zu ermöglichen (siehe Abschnitt 4.2). Zuletzt wird beschrieben wie die wichtigsten Techniken der einzelnen Komponenten realisiert wurden (siehe Abschnitt 4.3).

4.1 Verwendete Bibliotheken

JMonkeyEngine Für die dreidimensionale Visualisierung des modellierten Systems wird die freie 3D-Engine JMonkeyEngine[1] verwendet. Sie ist speziell für Java-Spiele-Entwickler ausgelegt und ist daher in der Lage, die Darstellung vieler Objekte performant umzusetzen. Die Bibliothek ist komplett in Java realisiert und lässt sich einfach in Desktop Anwendungen integrieren. Für die Entwicklung stehen eine umfangreiche Dokumentation und zahlreiche Beispielanwendungen zur Verfügung.

JFreeChart Zur Visualisierung von Messwerten und Ergebnissen wurde die freie Open Source Bibliothek JFreeChart[2] verwendet. Damit können Diagramme erstellt werden, die sich in die Entwicklungsumgebung integrieren lassen. Über mitgelieferte Funktionen können Diagramme einfach und übersichtlich skaliert werden, um so viele Messwerte detailliert betrachten zu können. Die Bibliothek wurde vollständig in Java umgesetzt und kann plattformübergreifend verwendet werden.

JDOM Für das Laden und Speichern von Objekten in ein XML Format wurde die freie Bibliothek JDOM[3] verwendet. JDOM ist optimiert für das Verarbeiten großer Datenmengen und benötigt gleichzeitig geringe Speicherressourcen gegenüber vergleichbaren Bibliotheken.

[1] http://www.jmonkeyengine.org/
[2] http://www.jfree.org
[3] http://www.jdom.org

4.2 Realisierung der Entwicklungsumgebung

Die Entwicklungsumgebung wurde in mehrere Komponenten untergliedert. In diesem Abschnitt werden wichtige Techniken beschrieben, die für die Realisierung aller Komponenten aus Abschnitt 3.3 zum Einsatz kamen. Dabei wird zuerst beschrieben wie eine Komponente unterteilt wurde um die Austauschbarkeit und Erweiterbarkeit zu ermöglichen (siehe Abschnitt 4.2.1). Anschließend werden die Techniken der Model-View-Control Unterteilung der Komponenten beschrieben (siehe Abschnitte 4.2.2, 4.2.3 und 4.2.3). Darauf aufbauend wird beschrieben wie die Interaktion der Teilkomponenten View und Model realisiert wurde (siehe Abschnitt 4.2.5). Zuletzt wird beschrieben wie der Zustand der Entwicklungsumgebung gespeichert und wiederhergestellt wird (siehe Abschnitt 4.2.6).

4.2.1 Plug-Ins

Die Entwicklungsumgebung wurde in mehrere Plug-Ins unterteilt, damit jeweilige Funktionalitäten gekapselt werden. Die Untergliederung stärkt das Separation of Concerns Prinzip. Um einen flexiblen Programmentwurf zu ermöglichen, der eine spätere Änderung oder Erweiterung erleichtert und eine Wiederverwendbarkeit, wurden die Komponenten nach dem Model-View-Control-Pattern gegliedert (siehe Abschnitt 3.3). Die Komponenten (Environment, Simulation, Distributor, Analysis und Monitor) wurden jeweils in ein Model- und View-Plug-In aufgeteilt. Das Model-Plug-In enthält die darzustellenden Daten und die Geschäftslogik. Das View-Plug-In enthält die Präsentation und Steuerung. Damit die verwendeten Bibliotheken in der gesamten Anwendung verfügbar sind werden auch diese als separates Plug-In implementiert. Dies ermöglicht auch ein leichteres Aktualisieren der Bibliotheken. Kern der Anwendung bildet das Platform-Plug-In, welches alle anderen Plug-Ins lädt und verwaltet (siehe Abbildung 4.1). Die View-Plug-Ins werden als RCP-Plug-In implementiert und erhalten neben einer

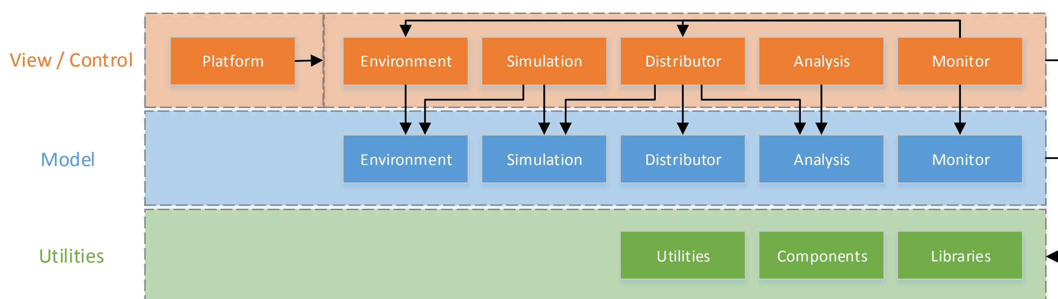

Abbildung 4.1: Übersicht der Systemkomponenten

Aktivator-Klasse, die den Lebenszyklus des Plug-Ins steuert, noch zusätzliche Advisor-Klassen für die Verwaltung der Workbench und deren Komponenten. Die Model-Plug-Ins werden als OSGi-Plug-Ins implementiert. Diese enthalten lediglich die Aktivator-Klasse, da keine Workbench und andere GUI Elemente benötigt werden.

Abhängigkeiten der Plug-Ins Damit ein Plug-In verwendet werden kann, muss festgelegt werden, welche Pakete exportiert und welche importiert werden müssen. Bei importierten Paketen handelt es sich um Pakete anderer Plug-Ins, die verwendet werden. Mit den exportierten Paketen lässt sich festlegen, welche Klassen für andere Plug-Ins sichtbar sein sollen, die dieses Plug-In verwenden wollen. Bei den Model-Plug-Ins wird die Paketstruktur daher so gegliedert, dass alle Interfaces, die gleichzeitig auch die Schnittstellen des Plug-Ins bilden, in einem Paket namens API liegen. So kann dieses Paket für andere Plug-Ins exportiert werden, ohne Teile der Geschäftslogik direkt freizugeben.

Extensionpoints und Extensions Eine der zentralen Technologien von Eclipse und von Eclipse-basierten Rich-Client-Anwendungen ist der Mechanismus von Erweiterungspunkten (extension points) und Erweiterungen (extensions). Mithilfe dieses Mechanismus lassen sich Funktionalitäten von bestehenden Plug-Ins erweitern. Dieser Ansatz wird verwendet, um die Interaktionen zwischen Model und View zu realisieren.

4.2.2 Model

Die Model-Plug-Ins wurden mit einer einheitlichen Schnittstelle implementiert. Über die DataFactory Klasse des Models wird der Zugriffspunkt der Daten bereitgestellt. Mit der Operation *getInstance()* wird eine Singleton Instanz des DataHandlers erstellt, damit bei jedem weiteren Aufruf auf der gleichen Objekt Instanz gearbeitet wird. Der DataHandler verwaltet alle einzelnen Datenobjekte und bietet eine konsistente Schnittstelle zum Hinzufügen und Entfernen von Daten und Model-Listener.

```
1  public interface IDataHandler
2  {
3      public IData createData( String filePath );
4
5      public void addData( IData data );
6      public void addDatum( List<String> filePathList );
7      public void removeData( IData data );
8
9      public void addModelChangeListener(
10         IDatumChangeListener datumChangeListener );
```

```
11      public void removeModelChangeListener(
12          IDatumChangeListener datumChangeListener );
13
14      public List<IData> getData();
15      public List<IData> getData( String name );
16  }
```

Listing 4.1: Interface IDataHandler für die Datenverarbeitung im Model

Data Handling Damit ein Datenobjekt angelegt werden kann, wird ein Dateipfad benötigt, der zur eindeutigen Identifizierung dient sowie den Speicherort definiert. Über den DataHandler kann mit der Operation *createData(filePath)* ein Datenobjekt erzeugt werden. Dieses Objekt wird mit der Operation *addData(Data)* dem DataHandler hinzugefügt. Nun kann mit der DataFactory über die Model-Schnittstelle darauf zugegriffen werden (siehe Abbildung 4.2).

Um einheitliche Zugriffe auf die Datenobjekte zu gewährleisten, sind einheitliche Operationen zum Laden und Speichern implementiert. Das Abspeichern erfolgt mit der JDOM-Bibliothek (siehe Abschnitt 4.1). Beim Aufruf der *save()*-Operation eines Datenobjekts werden alle Attribute, die zur Rekonstruktion des Objektzustands benötigt werden, in eine XML-Struktur serialisiert. Anschließend wird die Struktur im ASCII-Format unter dem Dateipfad des Objekts abgespeichert. Damit wird ein Überarbeiten der erstellten Datenobjekte auch außerhalb der Umgebung durch externe Werkzeuge ermöglicht.

Das Laden eines Datenobjekts erfolgt ebenfalls über den DataHandler. Mit der Operation *createData(filePath)* wird zuerst ein leeres Datenobjekt angelegt, welches den Pfad der zu ladenden Datei beinhaltet. Anschließend kann über die Operation *load()* des erzeugten Objekts die XML-Struktur deserialisiert werden. Dies erfolgt ebenfalls über die Funktionalität der JDOM-Bibliothek. Damit das Datenobjekt dem DataHandler bekannt ist, muss es wie beim Anlegen eines neuen Objekts auch mit der *addData(Data)*-Operation hinzugefügt werden.

ChangeListener Um Änderungen des Models auf die View oder andere Komponenten übertragen zu können, wurden zwei Arten von ChangeListenern implementiert.

Der DataHandler ermöglicht es, ChangeListener zu registrieren (im Folgenden DataHandlerListener genannt), die auf Änderungen des DataHandlers reagieren. So können Ereignisse überwacht werden, die beim Hinzufügen oder Entfernen von Datenobjekten ausgelöst werden.

Jedes Datenobjekt bietet ebenfalls die Möglichkeit ChangeListener zu registrieren (im Folgenden DataListener genannt), die auf Ereignisse des jeweiligen Datenobjekts reagieren. Während der DataHandlerListener nur auf strukturelle Änderungen reagiert, bietet der DataListener

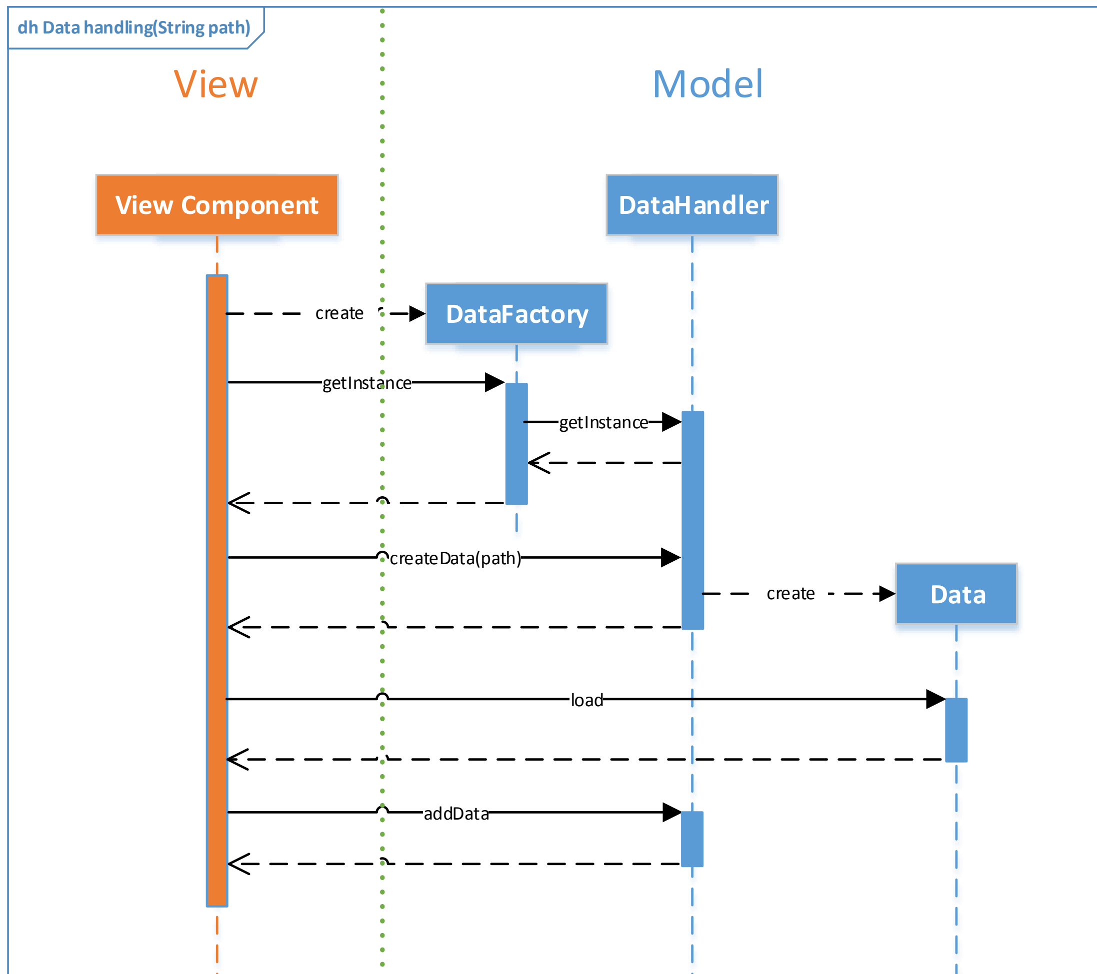

Abbildung 4.2: Hinzufügen von Datenobjekten

die Möglichkeit, konkrete Event-Typen zu erzeugen, auf die reagiert werden sollen. Über die Schnittstelle des Datenobjekts wird mit der Operation *addChangeListener(...)* der Listener sowie eine Klasse des Event-Typen übergeben, auf den reagiert werden soll.

```
@Override
public void addModelChangeListener(
    IEnvironmentsChangeListener listener )
{
    this.listeners.add( listener );

}
```

Listing 4.2: Hinzufügen eines DataHandlerListener

```
private void fireEvent( IEnvironmentsChangeEvent event )
{
    for( IEnvironmentsChangeListener listener : this.listeners )
    { listener.changed( event ); }

}
```

Listing 4.3: Ereignissignalisierung in der Model Komponente

Zur Verwaltung der registrierten ChangeListener wurde eine HashMap implementiert, die als Schlüsselattribut die Event-Klasse beinhaltet. Als Werteattribut wird eine Liste von ChangeListenern übergeben. Wenn ein neuer ChangeListener registriert werden soll, wird überprüft, ob bereits eine Liste von ChangeListenern für diesen Event-Typen vorhanden ist. Ist bisher keine Liste verfügbar, wird der HashMap die Event Klasse und eine neue Liste übergeben. Anschließend wird der Listener der Liste hinzugefügt.

```
@Override
public void addChangeListener(
    IEnvironmentChangeListener listener,
    Class<? extends IEnvironmentChangeEvent> eventClass )
{
    if( !this.listener.containsKey( eventClass ) )
    {
        this.listener.put(
            eventClass,
            new LinkedList<IEnvironmentChangeListener>() );
    }

    this.listener.get( eventClass ).add( listener );

```

```
15 }
```

Listing 4.4: Hinzufügen eines DataListener

Die Events werden in hierarchische Strukturen aufgebaut. Beim Environment wurden die Events unterteilt in das allgemeine ChangeEvent, das jegliche Änderungen signalisiert. Daraus abgeleitet sind das AtomChangeEvent und das GroupChangeEvent, die jeweils Änderungen der Objekte des Environments signalisieren. Für die Atom-Objekte des Environments wurde das AtomChangeEvent weiter unterteilt, um verschiedene Eigenschaftsänderungen zu signalisieren.

Innerhalb des Datenobjekts werden Ereignisse über die Operation *fireEvent(Event)* ausgelöst. Dabei wird das spezifische Event übergeben. Durch den hierarchischen Aufbau der Ereignisse können Listener in verschiedenen Abstraktionsstufen registriert werden. So werden bei einem AtomChangeEvent auch alle Listener benachrichtigt, die sich auf das ChangeEvent registriert haben. Wenn ein ChangeEvent ausgelöst wird, werden nur Listener informiert, die sich für dieses Event registriert haben. Die Listener, die sich konkret für das AtomChangeEvent registrieren, werden in diesem Fall nicht benachrichtigt.

```
1  public void fireEvent( IEnvironmentChangeEvent event )
2  {
3      for( Class<? extends IEnvironmentChangeEvent> eventIt :
4          this.listener.keySet() )
5      {
6          if( eventIt.isInstance( event ) )
7          {
8              for( IEnvironmentChangeListener listener :
9                  this.listener.get( eventIt ) )
10             { listener.change( event ); }
11
12         }
13
14     }
15
16 }
```

Listing 4.5: Ereignissignalisierung in dem Datenobjekt

4.2.3 View

Data-Binding Ein User-Interface besteht in RCP-Anwendungen aus Elementen (sogenannten Widgets), die bestimmte Eigenschaften von Objekten des Models abbilden. Damit Änderun-

gen im Modell sowie im Widget synchron sind, ist es hilfreich sich einer Technik zu bedienen, die dieses weitgehend automatisiert. JFace bietet die Technik des Data-Binding für alle zur Verfügung gestellten Widgets.

Damit ein Objekt gebunden werden kann, muss es in der Lage sein, PropertyChangeListener zu verwalten und bei Änderungen die entsprechenden PropertyChangeEvents auszulösen. Um dies zu erreichen, wird jedem Objekt eine PropertyChangeSupport Instanz hinzugefügt. Darüber hinaus werden zusätzlich die Operationen *addPropertyChangeListener()* und *remove-PropertyChangeListener()* benötigt.

```
1  private PropertyChangeSupport changes;
2
3  public Atom(...)
4  {
5      ...
6      this.changes = new PropertyChangeSupport( this );
7      ...
8  }
9
10 public void addPropertyChangeListener(
11     PropertyChangeListener listener )
12 { changes.addPropertyChangeListener( listener ); }
13
14 public void removePropertyChangeListener(
15     PropertyChangeListener listener )
16 { changes.removePropertyChangeListener( listener ); }
```

Listing 4.6: Vorausgesetzte Operationen und Attribute für das DataBinding

Im Folgenden wird das Binding zwischen einem Model Objekt *observer* und einem JFace Widget *textName* dargestellt. Um ein Binding zu erstellen, wird ein DataBindingContext-Objekt erzeugt, welches die entsprechenden Listener bei dem Binding-Partner registriert. Das DataBindingContext-Objekt benötigt die beiden Binding-Partner als IObservableValue-Objekt. Dieses beinhaltet eine Referenz des zu bindenden Objekts, sowie das Attribut, das geändert oder überwacht werden soll. Anschließend wird das Binding mit einer Strategie versehen, die angibt wann ein Update der Informationen bei welchem Binding-Partner vollzogen werden soll. Dies geschieht über die UpdateValueStrategy. Im folgenden Code Abschnitt ist die Strategie mit dem Attribut $POLICY_UPDATE$ gezeigt. Durch dieses wird das Widget bei jeder Änderung des Model-Objekts aktualisiert. Durch das Attribut $POLICY_ON_REQUEST$ übernimmt das Model-Objekt die Änderung des Widgets nur wenn es gewünscht ist. Zusätzlich kann jede Strategie durch einen Validator verfeinert werden. Mit dem *setAfterGetValidator* wird der Wert

erst übertragen, wenn der zu übernehmende Wert vom Validator akzeptiert wurde. In diesem Fall wird erst der neue Wert des Widgets angefordert, anschließend im Validator geprüft und bei Akzeptanz an das Model-Objekt übertragen.

```
DataBindingContext nameBindingContext = new DataBindingContext();
IObservableValue nameModelObservable =
    BeanProperties.value( "name" ).observe( observer );
IObservableValue nameViewObservable =
    WidgetProperties.text( SWT.Modify ).observe( textName );

UpdateValueStrategy modelToView =
    new UpdateValueStrategy( UpdateValueStrategy.POLICY_UPDATE );
UpdateValueStrategy viewToModel =
    new UpdateValueStrategy( UpdateValueStrategy.POLICY_ON_REQUEST );
viewToModel.setAfterGetValidator( new EnvironmentNameValidator() );

nameBindingContext.bindValue(
    nameViewObservable, nameModelObservable,
    viewToModel, modelToView );
```

Listing 4.7: Erzeugen eines Bindings zwischen View und Model

Durch die $POLICY_ON_REQUEST$ Strategie muss eine Synchronisation zwischen den Eingaben der View und den Objekten des Models mit der *updateModels()*-Operation des DataBindingContext-Objekt gestartet werden. Die Synchronisation kann wie im Folgenden gezeigt durch Benutzer Interaktionen ausgelöst werden.

```
textName.addFocusListener( new FocusListener()
{
    @Override
    public void focusLost( FocusEvent e )
    { nameBindingContext.updateModels(); }

    ...

} );
```

Listing 4.8: Aktualisierung eines Bindings durch Benutzerinteraktion

SelectionService Der SelectionService stellt ein zentrales Konzept der Kommunikation von RCP-Komponenten dar. So können Views über den Service miteinander Kommunizieren und Daten austauschen. Auch ActionHandler, Wizards, Dialoge und andere Komponenten

können sich an Informationen aus dem SelectionService bedienen. Ein JFace-Widget kann beim SelectionService registriert werden. Über das global zur Verfügung stehende Interface des Services kann aus jedem Bereich der RCP-Anwendung abgefragt werden, welches Objekt aktuell ausgewählt ist. Auch Textfelder können bei dem Service registriert werden, um Auskunft über das ausgewählte Textsegment zu geben. Der SelectionService kann in zwei Geltungsbereichen verwendet werden. Wenn nur die Auswahl einer bestimmten View beobachtet werden soll, kann der spezifische Service eines Viewparts verwendet werden. Soll allerdings auf jegliche Selektionen reagiert werden, kann der Workbench SelectionService verwendet werden, der alle Ereignisse weiterleitet. Bei der Anmeldung eines Widgets (welches Ereignisse bereitstellt) im SelectionService muss kein Geltungsbereich ausgewählt werden.

Wizards Zur Erzeugung der einzelnen Model-Objekte wurden Wizard-Widgets implementiert. Ein Wizard wird über die Extension *org.eclipse.ui.newWizards* deklariert. Als Attribut wird eine Klasse vom Typ *IWizard* benötigt sowie eine Extension-Id und ein Anzeigename. Das Instantiieren erfolgt über die *ExtensionRegistry*. Dort sind alle Extensions hinterlegt und können über die Extension-Id abgerufen werden. Über die Operation *getExtensionPoint(String)* der Registry werden alle Extensions abgerufen, die den Extension Point *org.eclipse.ui.newWizards* erweitern. Mithilfe der eindeutigen Extension-Id können die erhaltenen Extensions gefiltert werden, um den gewünschten Wizard zu erhalten.

Für die Implementierung des Wizards wird die *org.eclipse.jface.wizard.Wizard* erweitert. Hierzu werden drei Operationen benötigt. Die *addPages()*-Operation wird beim Instantiieren des Wizards aufgerufen. Dort werden die einzelnen WizardPages dem Wizard hinzugefügt. Die Operation *canFinish()* gibt an, ob der Dialog über den Finish Button fertiggestellt werden kann. Ob der Wizard fertiggestellt werden kann hängt von den einzelnen Pages ab. Diese haben einen eigenen Zustand, der erfragt werden kann. Die logische Verknüpfung aller Page-Zustände bestimmt, ob ein Wizard abgeschlossen werden darf. Sobald der Finish Button gedrückt wurde, wird die Operation *performFinish()* ausgeführt. An dieser Stelle erfolgt die Erzeugung des Model-Objekts.

4.2.4 Control

Wizard State In der Anwendung werden mehrere Komponenten mit einem Setup-Wizard-Dialog erstellt. So wird ermöglicht, umfangreiche Einstellungen vorzunehmen, bevor die Komponente erzeugt wird. In einer RCP-Anwendung werden die Wizards von JFace bereitgestellt. Dadurch kann ein Setup einer Komponente schnell und mit wenig Code implementiert werden. Der Rohaufbau eines Wizards unterteilt sich in den Wizard-Dialog, der Funktionalität

für das Fenster und die Buttons Cancel, Back, Next und Finish bereitstellt. Gefüllt wird der Dialog mit einer oder mehreren WizardPages, die jeweils mit den Back- und Next-Buttons angesteuert werden. Über den Finish-Button werden Operationen zum Erstellen der Komponente aufgerufen und der Dialog geschlossen. Mit dem Cancel-Button wird der Dialog geschlossen ohne eine Komponente zu erzeugen.

Für die Erzeugung einer Simulationskomponente wurde ein Wizard in zwei Wizard-Pages unterteilt. Auf der ersten Seite wurde ein Textfeld implementiert, um der Komponente einen Namen zuzuweisen. Um der erstellten Komponente einen Speicherort zuzuweisen, wurde ein weiteres Textfeld implementiert, das mit einem voreingestellten Ordnerpfad befüllt ist. Im Weiteren folgen zwei Textfelder zur Einstellung der auszuführenden Simulationsschritte und der Zeit, die zwischen den einzelnen Schritten gewartet werden soll. Auf der zweiten Wizard-Page muss der Simulation ein bereits existierendes Environment zugewiesen werden, mit dem die Simulation ausgeführt werden soll. Erst nach korrekter Einstellung all dieser Attribute kann eine Simulationskomponente über den Finish-Button angelegt werden. JFace bietet mit dem Wizard-Widget die Möglichkeit den Status einer Page zu setzten. Mit der Wizard-Page-Operation *setPageComplete(Boolean)* kann dem Wizard-Dialog mitgeteilt werden welche Buttons aktiv sowie inaktiv sein sollen. Damit der Zustand der Buttons von den einzelnen Attribut-Werten abhängig ist wurde ein WizardStateHandler implementiert, der den Next- und Finish-Button deaktiviert, wenn nicht alle Attribute korrekt eingegeben wurden. Des Weiteren setzt der WizardStateHandler den Infotext der Wizard-Page, welcher Aufschluss über die nächsten Schritte gibt.

Für jedes Attribut der Wizard-Page, das zur Erzeugung der Komponente benötigt wird, werden *IStateFaulureCondition*-Objekte erstellt. Diese überprüfen den Wert des Attributs auf Akzeptanz und geben im Fehlerfall eine spezifische Fehlermeldung zurück. Im folgenden Code Abschnitt sind zwei Bedingungen gezeigt, die den Wert des Simulationspfades überprüfen. In der ersten Bedingung wird überprüft, ob der Standard-Pfad ausgewählt ist. Wenn nicht wird überprüft, dass der angegebene Pfad nicht leer ist. In der zweiten Bedingung wird im Falle eines benutzerdefinierten Pfades überprüft, ob der Pfad existiert und konform ist. Jede Bedingung liefert im Fehlerfall, mit der Operation *getFailureMessage()*, eine problemorientierte Meldung zurück.

```
1 conditionEmptyLocation = new IStateFailureCondition()
2 {
3     @Override
4     public String getFailureMessage()
5     { return ...; }
6 
```

```
7       @Override
8       public boolean failuer()
9       { return btnUseDefaultLocation.getSelection() ||
10               !( txtLocation.getText() == null ||
11                  txtLocation.getText().equals( "" ) ); }
12
13  };
14
15  conditionIllegalLocation = new IStateFailureCondition()
16  {
17      @Override
18      public String getFailureMessage()
19      { return ...; }
20
21      @Override
22      public boolean failuer()
23      { return btnUseDefaultLocation.getSelection() ||
24               new File( txtLocation.getText() ).isDirectory(); }
25
26  };
```

Listing 4.9: Fehlerbedingungen des WizardStateHandlers

Der WizardStateHandler wird innerhalb einer Wizard-Page instantiiert und bekommt als Argumente die Page-Instanz, um Zugriff auf die Operation *setPageComplete(Boolean)* zu erhalten, sowie die Meldung des fehlerfreien Zustands der Wizard-Page übergeben. Dieser wird als Infotext im Wizard angezeigt. Mit der *addStateCondition(IStateFailureCondition)*-Operation werden die Bedingungen dem WizardStateHandler hinzugefügt.

```
1  wizardStateHandler = Utilities.createWizardStateHandler(
2          this,
3          "Kein Fehler" );
4
5  wizardStateHandler.addStateCondition( conditionEmptyLocation );
6  wizardStateHandler.addStateCondition( conditionIllegalLocation );
```

Listing 4.10: Instantiierung und Initialisierung des WizardStateHandlers

Bei Interaktion des Benutzers mit dem User Interface erfolgt die Prüfung der einzelnen Bedingungen. Sobald ein Attribut geändert wird, kann in dem Input-Element auf die Änderung reagiert werden. Die Prüfung der entsprechenden Bedingung wird mit der Operation *check(IStateFailureCondition)* in dem entsprechenden ModifyListener gestartet (siehe Abbildung 4.3). Der WizardStateHandler speichert die einzelnen Bedingungen als Liste. Das Argument der

Operation ist die erste Bedingung, die überprüft wird. Anschließend wird die Liste zyklisch durchlaufen und alle weiteren Bedingungen überprüft, bis eine Bedingung einen Fehler zurückgibt oder das übergebene Argument wieder erreicht wird. Dieser Ansatz hat den Vorteil, dass Fehler direkt dort wo sie entstehen überprüft werden. Auch Fehler, die aus mehreren Attributen entstehen, werden mit dieser Methode aufgedeckt. So kann bei der Eingabe eines Namens und des zugehörigen Dateipfades bei jeder der beiden Eingaben überprüft werden, ob die Datei bereits existiert. Im folgenden Code Abschnitt wird ein ModifyListener einem Textfeld hinzugefügt, der dafür sorgt, dass bei jeder Eingabe die zugehörige Bedingung geprüft wird. Die zweite Bedingung für dieses Textfeld wird erst anschließend überprüft, da sie im nachfolgenden Listenplatz des WizardStateHandlers hinzugefügt wurde.

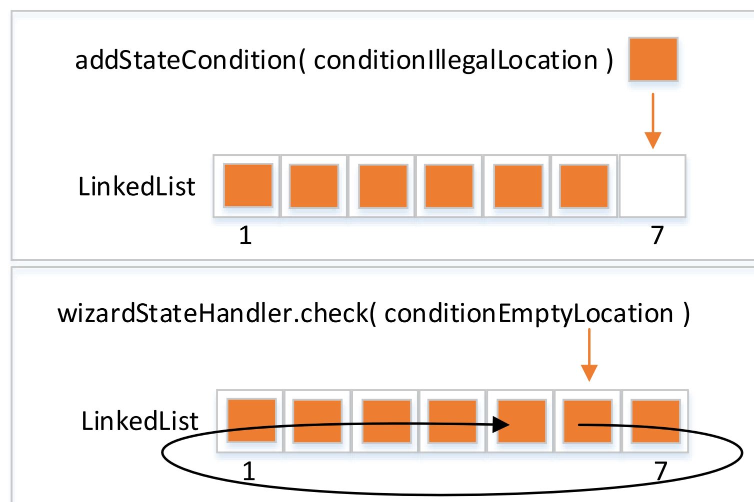

Abbildung 4.3: Arbeitsweise des WizardStateHandlers. In Abhängigkeit des Eingabeelements werden die Fehlerbedingungen durchlaufen und überprüft

```
txtLocation.addModifyListener( new ModifyListener()
{
    @Override
    public void modifyText( ModifyEvent e )
    { wizardStateHandler.check( conditionEmptyLocation ); }

} );
```

Listing 4.11: Fehlerprüfung des WizardStateHandlers durch Benutzerinteraktionen

Selektionen In der Anwendung wurden Dialogfenster implementiert, um die Objekterzeugung zu vereinfachen. Zum Erstellen eines Simulationsobjekts wurde ein Dialog verwendet, der ein zugehöriges Environment-Objekt der Simulation zuordnet. Bei den Simulationsaktionen wurde ein Dialog verwendet, um einzelne Atome oder Gruppen des Environments auszuwählen. Auch weitere Komponenten der Anwendung wurden mittels Dialogfenstern realisiert. Damit ein Dialog entworfen werden konnte, welcher für die verschiedenen Aufgaben einsetzbar ist, wurde eine Methode realisiert, die unterschiedliche Auswahlmöglichkeiten zulässt. Somit können unterschiedliche Anforderungen umgesetzt werden. Das Interface *IDialogSelectionRestriction* weißt einem Dialog zulässige Auswahloptionen zu. Über die Interface-Operation *restriction(List<Object> selection)* teilt ein Dialog mit, welche Objekte ausgewählt sind. Die konkrete Implementierung des Interfaces kann dadurch eine kontextabhängige Prüfmethode realisieren. Über den Rückgabewert wird dem Dialog mitgeteilt, ob die Auswahl akzeptiert wird. Demnach können die Steuerungselemente für das Abbrechen oder Fortfahren des Dialogs entsprechend aktiviert werden. In folgendem Code-Abschnitt ist eine *IDialogSelectionRestriction* gezeigt. Dadurch kann der Dialog, dem die Restriktion zugewiesen wurde, nur Auswahloptionen des Typs IEnvironment akzeptieren.

```
1  dialog.setSelectionRestriction( new IDialogSelectionRestriction()
2  {
3
4      @Override
5      public boolean restriction( List<Object> selection )
6      {
7          for( Object object : selection )
8          if( !( object instanceof IEnvironment ) ){ return false; }
9
10         return true;
11
12     }
13
14 } );
```

Listing 4.12: SelectionRestriction für die Auswahlbeschränkung von Dialogfenstern

Aktionen und Kommandos Seit Eclipse 3.3 steht mit dem Command-Framework ein einheitlicher Mechanismus zur Verfügung, Kommandos zu deklarieren und diese in Menüs oder Toolbars einzufügen. Ein Kommando definiert lediglich eine ausführbare Aktion, nicht jedoch das Verhalten. Durch die deklarative Herangehensweise wird eine höhere Flexibilität und Erweiterbarkeit erreicht. So können Plug-Ins bestehende Menüstrukturen erweitern oder

zu einem vorhanden Kommando ein neues Verhalten hinzufügen. Ein weiterer Vorteil der Kommandos besteht darin, dass Menüs und Toolbars von der Anwendung angezeigt werden können, ohne Klassen aus dem jeweiligen Plug-In zu laden (vgl. Ebert (2011)). Die Deklaration eines Kommandos erfolgt über die RCP-Extensions in der Plugin.xml-Datei (siehe Abschnitt 2.3.2).

```
<extension point="org.eclipse.ui.commands">
    <command
        id="environment.command.openenvironment"
        name="Open Environment">
    </command>
</extension>
```

Listing 4.13: Kommando Deklaration in der Plugin.xml

Über den Extension Point *org.eclipse.ui.commandImages* kann einem Kommando eine Bilddatei zugewiesen werden, die in Menüs oder Toolbars angezeigt werden soll.

```
<extension
         point="org.eclipse.ui.commandImages">
  <image
    commandId="environment.command.openenvironment"
    icon="icons/environment_open.gif">
  </image>
</extension>
```

Listing 4.14: Image Deklaration für Commands

Mit dem Extension Point *org.eclipse.ui.menus* kann ein Kommando einem Menü zugewiesen werden. Hierzu wird ein *menuContribution*-Handler benötigt, der festlegt, an welchen Viewpart das Menü gebunden werden soll. Über Schlüsselwörter wie *popup* oder *toolbar* kann zur Laufzeit ermittelt werden, welchem Element eines Viewparts das Menü zugeordnet werden soll. Jeder Viewpart verfügt standardmäßig über ein Toolbar-Element. Das Popup-Menü muss dagegen an ein JFace-Widget gebunden werden, das sich innerhalb eines Viewparts befinden. Als Unterelement eines menuContribution-Handlers wird ein Kommando-Element deklariert, das über eine Id die zugehörige Kommando-Erweiterung referenziert. Optional kann ein Anzeigetext und das Erscheinungsbild deklariert werden, das in dem Menü angezeigt werden soll.

```
<extension point="org.eclipse.ui.menus">
  <menuContribution
    allPopups="false"
    locationURI="popup:environment.explorer">
```

```
5      <command
6        commandId="environment.command.openenvironment"
7        label="Open Environment"
8        style="push">
9      </command>
10   </menuContribution>
11 </extension>
```

Listing 4.15: Deklaration von Menüstrukturen in der Plugin.xml

Damit dem Kommando ein Ausführungsverhalten zugewiesen werden kann, wird die Extension *org.eclipse.ui.handlers* benötigt. Ein Handler bindet die Aktion eines Kommandos über dessen Id an eine vorhandene Klasse. Bei Betätigung des Kommandos wird die Handler-Klasse aus dem entsprechenden Plug-In instantiiert.

```
1 <extension point="org.eclipse.ui.handlers">
2   <handler
3     class="environment.control.handler.DefaultOpenEnvironmentHandler"
4     commandId="environment.command.openenvironment">
5   </handler>
6 </extension>
```

Listing 4.16: Zuweisung des Ausführungsverhaltens für einen Command

Die konkrete Handler-Klasse realisiert das Interface IHandler. Das Verhalten des Kommandos wird in der vom Interface bereitgestellten Operation *execute(...)* implementiert.

Manche Kommandos eines User-Interfaces sollen erst aktiviert werden, wenn ein bestimmtes Ereignis aufgetreten ist. So ist beispielsweise für das Entfernen eines Listenelements der Kommando erst aktiv, wenn ein Element ausgewählt ist. Um dieses Verhalten zu realisieren, wird die Extension *org.eclipse.ui.services* verwendet. Über die Services-Erweiterung lassen sich sogenannte Source-Provider deklarieren, um die Sichtbarkeit von Kommandos zu bestimmen. Die Deklaration erfolgt ebenfalls über die Plugin.xml. Dazu wird eine Klasse angegeben, die den Source-Provider implementiert sowie eine Variable deklariert, die den Zustand des Kommandos enthält.

Eine Source-Provider-Klasse muss das Interface *ISourceProvider* realisieren. Die wichtigsten Operationen sind *getProvidedSourceNames()*, über die alle vorhandenen Variablen des Providers erfragt werden kann sowie *getCurrentState()*, mit der eine Map zurück gegeben wird, die alle Variablen und deren Werte beinhaltet. Damit bei Benutzerinteraktionen die Variablen des Providers aktualisiert werden, wird der Provider als ChangeListener an bestimmte Widgets übergeben. In der konkreten Realisierung wurde ebenfalls eine *ISelectionRestriction* (siehe Abschnitt 4.2.4) verwendet und an ein Widget gebunden. Im Falle einer Auswahländerung wird

das selektierte Objekt überprüft und demnach die Variable des Providers aktualisiert. Im folgenden Code-Abschnitt wird überprüft, welches Environment ausgewählt wurde. Anschließend wird die Änderung dem Source-Provider mitgeteilt.

```
this.slectionRestriction = new ISelectionRestriction()
{
  @Override
  public void restriction( Object object )
  {
    AddProvider.this.state = NOSELECTION;
    if( object instanceof TreeObject )
    {
      if( ( ( TreeObject ) object ).getObject() instanceof IEnvironment )
        AddProvider.this.state = ENVIRONMENT;
      else if( ( ( TreeObject ) object ).getObject() instanceof IGroup )
        AddProvider.this.state = GROUP;
      else if( ( ( TreeObject ) object ).getObject() instanceof IAtom )
        AddProvider.this.state = ATOM;

      AddProvider.this.fireSourceChanged( ISources.WORKBENCH, NAME, state );

    }

  }

};

Utilities.registerSelectionListener( this.slectionRestriction );
```

Listing 4.17: SelectionRestriction zur Aktivierung von Commands

Die Aktivität des Kommandos und dessen Abhängigkeit zur Variable wird ebenfalls über den Extension-Mechanismus deklarativ beschrieben. Einem Kommando kann das Element *VisibleWhen* angehängt werden, womit komplexe Abhängigkeiten zwischen Variablen definiert werden. Über Logik-Elemente wie *and*, *or* und *not* können verschiedene Variablen überprüft werden sowie eine Variable auf verschiedene Werte. Im folgenden Code Abschnitt wird das Sichtbarkeitsverhalten des Kommandos zum Hinzufügen einer Environment-Gruppe deklariert. Dieses Verhalten ist von der Selektion der Environment-Komponente abhängig. Mit dem *with*-Element wird die Variable des Providers angegeben, in der die Informationen der aktuellen Auswahl enthalten sind. Eine Gruppe kann nur hinzugefügt werden, falls entweder ein Environment oder eine Gruppen-Komponente ausgewählt wurde. Dieses Verhalten wird

über das Logik-Element *or* deklariert. Mithilfe des *equals* Elements wird die Variable auf den entsprechenden Wert überprüft.

```
<command
  commandId="environment.command.addgroup"
  label="Add_Group"
  style="push">
  <visibleWhen checkEnabled="false">
    <with
      variable="environment.command.active">
      <or>
        <equals
          value="GROUP">
        </equals>
        <equals
          value="ENVIRONMENT">
        </equals>
      </or>
    </with>
  </visibleWhen>
</command>
```

Listing 4.18: Regeldeklaration für die Sichtbarkeit eines Kommandos

4.2.5 View-Model Interaktion

Über die Wizards der View können Datenobjekte erzeugt werden, die in dem zugehörigen Model abgelegt und verarbeitet werden (siehe Abbildung 4.4). Da die Model-View-Komponenten als eigenständige Plug-Ins implementiert wurden, ist es nötig einen Interaktions-Mechanismus zu implementieren, der die Austauschbarkeit der View weiterhin gewährleistet. Mit diesem Aspekt wurden die RCP-Extensions und Extension-Points verwendet.

Die Model-Komponente deklariert in der Plugin.xml einen Extension Point, der das Interface IDataFactory realisieren muss. Die konkrete Realisierung des Interfaces wurde ebenfalls im Model als DataFactory implementiert. Die DataFactory stellt die zentrale Schnittstelle des Plug-Ins dar. Alle Zugriffe auf die Model-Komponente erfolgen über diese Schnittstelle. Die DataFactory liefert eine Singleton-Instanz des DataHandlers (siehe Abschnitt 4.2.2).

In der View-Komponente wird das Model-Plug-In als Required-Plug-In deklariert. Dadurch wird festgelegt, dass dieses Plug-In nicht ohne das zugehörige Model ausgeführt werden kann. Dennoch ist ein Austausch des Model-Plug-Ins weiterhin möglich. Die Deklaration erfolgt ebenfalls über die Plugin.xml. Sobald diese Abhängigkeit definiert wurde, stehen der View-

Komponente alle Extension-Points des Models zur Verfügung. Der angelegte Extension-Point für die IDataFactory kann daraufhin vom View-Plug-In als Extension verwendet werden. Dazu wird eine konkrete Realisierung des Extension-Interfaces IDataFactory benötigt. Die Realisierung wird von dem Model mit der Klasse DataFactory zur Verfügung gestellt. Der Zugriff auf die Datenobjekte des Models erfolgt nun in den Widgets über die Extension-Registry. Da bei jedem Abruf aus der Registry eine neue Instanz einer Extension erstellt wird, kommt die DataFactory zum Einsatz. Diese gibt eine Singleton-Instanz des DataHandlers zurück.

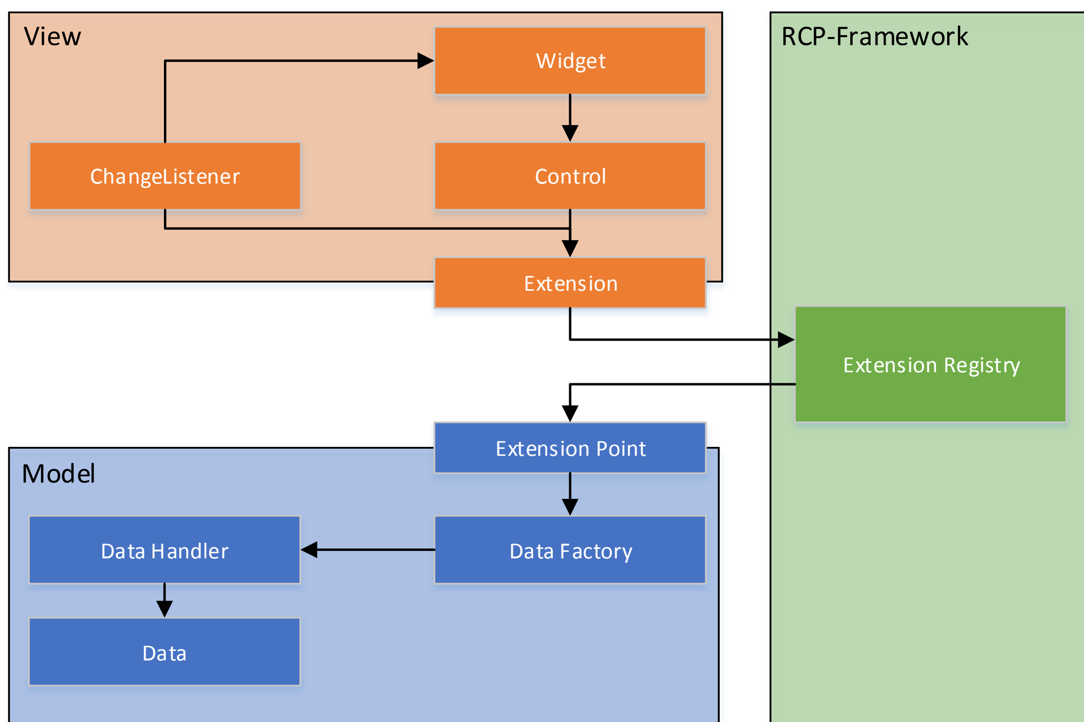

Abbildung 4.4: Interaktion zwischen View und Model Plug-Ins

4.2.6 State Recovery

Wem der Umgang mit Eclipse oder anderen RCP-Anwendungen vertraut ist, dem ist bekannt, dass nach einem Neustart der Anwendung an der gleichen Stelle weitergearbeitet werden kann, an der gestoppt wurde. Eclipse RCP bietet verschiedene Mechanismen zur Wiederherstellung von vorgenommenen Einstellungen, geöffneten Viewparts und geladenen Dateien.

Settings Recovery Benutzereinstellungen werden von der Rich-Client-Platform verwaltet und bereitgestellt. Dafür wird der PreferenceService verwendet, der von jedem Plug-In zugänglich ist. Die Präferenzen sind in sogenannte Geltungsbereiche (Scopes) unterteilt. Die für die Anwendung verwendeten Scopes sind *default* und *instance*. Darüber hinaus existieren noch weitere Scopes, die in dieser Arbeit nicht thematisiert werden. Der Default-Scope bezeichnet die Vorbelegung der Benutzereinstellungen, die nach dem ersten Start der Anwendung verfügbar sind. Unter dem Instance-Scope werden die benutzerdefinierten Einstellungen abgelegt, die nachträglich verändert wurden. Diese werden in einem .metadata-Ordner im Anwendungsordner abgelegt. Die Präferenzen sind in eine baumartige Struktur unterteilt. Jedes Plug-In erhält einen Ast vom Wurzelelement. An jeden dieser Äste können Präferenz-Seiten (Preference-Pages) angehängt werden, die wiederum weitere Seiten beinhalten können. Jeder Seite können einzelne Attribute angehängt werden. Diese sind als Tripel, bestehend aus Name, Wert und Vorbelegung hinterlegt.

Um Plug-In-spezifische Präferenzen zu erstellen, wird die *org.eclipse.ui.preferencePage* Extension verwendet. Unter dieser Extension können einzelne Page-Elemente deklariert werden. Zur Identifikation wird das Attribut Id benötigt, um die Page später im PreferenceService zu lokalisieren. Darüber hinaus wird ein Name benötigt, unter dem die Seite im Einstellungsdialog aufzufinden ist. Für die konkrete Implementierung wird eine Klasse angegeben, die das Interface *IWorkbenchPreferencePage* realisiert. Diese Klasse verfügt über eine *init()*-Operation, in der ein PreferenceStore der Seite zugewiesen wird. An dieser Stelle ist es auch möglich einen eigens implementierten Store zu verwenden, der dem Interface IPreferenceStore genügt. Damit jedoch jede Seite den Store verwendet, der von der Workbench zur Verfügung gestellt wird, kann über den jeweiligen Plug-In Activator oder der globalen *Platform* Klasse, der zentrale Store verwendet werden. Mit der Page-Operation *createFieldEditors()* werden entsprechende Field-Editoren der Seite angehängt. Diese Editoren repräsentieren die unterschiedlichen Eingabemöglichkeiten für die Benutzereinstellungen. So existieren Field-Editoren als Textfelder, Checkboxen, Listen und weitere UI-Elemente. Jeder Editor erhält eine eindeutige Id, mit der ein einzelnes Attribut im PreferenceService identifizierbar ist.

Die Initialisierung der Benutzereinstellungen erfolgt über die Deklaration der Extension *org.eclipse.core.runtime.preferences*. Diese Erweiterung erlaubt es, ein Initializer-Element zu deklarieren, das angibt, welche Klasse für die Initialisierung der Default-Werte verantwortlich ist. Die konkrete Klasse wird von der abstrakten Klasse *AbstractPreferenceInitializer* abgeleitet und stellt die Operation *initializeDefaultPreferences* bereit. Über ein globales Scope-Objekt kann der entsprechende PreferenceScope ausgewählt werden, der die initialen Werte enthalten soll. Dafür wird standardmäßig der Default-Scope verwendet, da dieser von der Workbench

für diese Zwecke passend konfiguriert wurde. Alternativ können an dieser Stelle auch eigene Scope-Implementierungen verwendet werden, die allerdings aufwendig konfiguriert werden müssten. Mithilfe der Plug-In-Id wird die entsprechende Präferenz aus der Baumstruktur des PrefenreceService angefordert. Anschließend kann über die eindeutige PreferenceId der Default-Wert geschrieben werden.

Sollte der Benutzer in dem Einstellungsdialog ein Attribut ändern, wird dieses von der Workbench automatisch in den Instance-Scope geschrieben. Nach Anwendungsstart wird bei Zugriff auf eine Präferenz automatisch der Instance-Scope durchsucht. Wenn diese keinen Suchtreffer zurückliefert, wird der Standardwert aus dem Default-Scope verwendet.

Perspective Recovery Nach einem Neustart einer RCP-Anwendung werden alle zuletzt sichtbaren Viewparts angezeigt. Um dieses zu erreichen wurde der RCP-interne SaveAndRestore-Mechanismus verwendet. Dieser wird über das Anwendungs-Plug-In, von dem die Anwendung gestartet wird, aktiviert. In der Klasse *ApplicationWorkbenchAdvisor*, die für jedes RCP Plug-In vorhanden ist, wird die Operation *initialize()* erweitert. Als Argument wird der *WorkbenchConfigurer* übergeben. Die Wiederherstellung der Views wird über dessen Operation *setSaveAndRestore(Boolean)* aktiviert. Des Weiteren muss darauf geachtet werden, dass jeder angelegte Viewpart in der Plugin.xml des entsprechenden Plug-Ins das restorable Flag mit dem Wert *true* belegt ist.

View State Recovery Damit die Benutzerinteraktionen der einzelnen Viewparts wiederhergestellt werden können, wird der Mechanismus der RCP Mementos verwendet. Die Zusandsinformationen der einzelnen Views werden in einer persistierbaren Hierarchie von Key-Value-Paaren abgelegt. Persistiert werden die Mementos von der Workbench, als robustes Format angelehnt an die XML Struktur. Um die Mementos zu verwenden, müssen die Operationen *saveState(IMemento)* und *init(IViewSite, IMemento)* eines Viewparts überschrieben werden. Wie auch beim Perspective Recovery muss der SaveAndRestore-Mechanismus der Workbench aktiviert werden.

Data Recovery Die Wiederherstellung der Datenobjekte des Models erfolgt bei Aktivierung der Plug-Ins über die Aktivator Klasse. Zu Beginn wird eine Klasse *ContentInitializer* mit der *init()*-Operation aufgerufen. Hier wird der Hauptteil der Logik zur Datenwiederherstellung abgehandelt. Über den *PreferenceService* wird der Ordnerpfad angefordert, in dem die Datenobjekte abgespeichert sind. Zuerst wird geprüft, ob ein benutzerdefinierter Pfad angegeben wurde. Wenn dies nicht der Fall ist, wird der voreingestellte Pfad vom PreferenceService angefordert.

Darüber hinaus kann es vorkommen, dass Datenobjekte geladen werden, die nicht im Standardverzeichnis liegen. Für diesen Fall existiert eine vordefinierte Id unter der benutzerdefinierte Dateien gespeichert und geladen werden können. Die Dateipfade werden in folgender Form im PreferenceStore abgelegt: {Key:Id,Value:"Pfad1;Pfad2;...PfadN"}. Anschließend werden die ermittelten Dateipfade aus dem Standardverzeichnis und die Dateipfade der benutzerdefinierten Dateien in einer Liste an das Model übergeben, um die Datenobjekte wiederherzustellen (siehe Abschnitt 4.2.2).

4.3 Realisierung der Komponenten

In diesem Abschnitt werden die fünf wichtigsten Komponenten beschrieben, die für die Durchführung der Simulationsstudie entwickelt wurden. Hierbei werden allerdings nur die wichtigsten Techniken der Komponenten beschrieben, da eine detailliertere Ausführung den Rahmen dieser Arbeit überschreiten würde.

4.3.1 Platform

Um der Anwendung einen Einstiegspunkt zu bieten, wird ein Platform-Plug-In realisiert. Alle weiteren Plug-Ins werden von dieser Komponente als Required-Plug-In deklariert. So entsteht eine lose Kopplung der einzelnen Komponenten und die Austauschbarkeit sowie Erweiterbarkeit wird gefördert. Beim Starten des Plug-Ins werden alle abhängigen Plug-Ins geladen. Das Laden erfolgt nach der Lazy-Initialization-Technik. Das bedeutet, dass alle Informationen aus den jeweiligen Plugin.xml-Dateien geladen werden, jedoch die einzelnen Klassen nur nach Bedarf instantiiert werden. Darüber hinaus ist die Platform-Komponente für das Setup der Workbench verantwortlich. Das bedeutet, dass die Menüstruktur mit grundlegenden Aktionen belegt wird: für das Anzeigen der vorhanden Viewparts, Perspektiven sowie der Zugang zum Einstellungsdialog. Die jeweiligen Unterpunkte der Menüs und des Einstellungsdialogs werden über den Extension-Mechanismus von eingebundenen Plug-Ins erweitert.

Die Platform-Komponente besteht lediglich aus einem View-Plug-In, da keinerlei Datenhaltung benötigt wird. Sie stellt den zentralen Einstiegspunkt der Anwendung dar und stellt eine Perspektive bereit, die zu Beginn angezeigt wird. In der Perspektive befinden sich zwei Viewparts. Die Project-Explorer-View bietet einen Überblick über alle erzeugten Artefakte der Simulationsstudie. Dafür wurde ein Widget verwendet, in dem alle erzeugten Datenobjekte (die geladenen Artefakte) der einzelnen Models angezeigt werden. Für die Präsentation wurde eine Baumstruktur (TreeViewer) gewählt, damit die Datenobjekte als Unterelemente ihrer Komponenten gegliedert werden. Die Welcome-View beinhaltet ein Browser-Widget, das die

URL *http://www.atom-simulation.com/howto/* lädt. Unter dieser Adresse ist die Dokumentation der Anwendung zu finden. So wird direkt beim ersten Start eine Schritt-für-Schritt Anleitung angezeigt.

4.3.2 Environment

Zur Modellierung der Simulations-Environments ist eine Perspektive *Environment* umgesetzt, in der zwei Viewparts angezeigt werden: Environment-Explorer und Vector-Explorer.

Vector-Explorer Die View dient zur Erzeugung und Verwaltung von Vektoren. Um einen Vektor zu erstellen ist es nötig, zuvor ein VectorSet-Objekt anzulegen. Diese Objekte dienen als Sammlung einzelner Vektoren und stellen eine Gruppierung dar. Zur Verwaltung der Objekte wurde die View in eine Toolbar und ein TreeViewer-Widget gegliedert. Die Toolbar ist mit Kommandos für das Hinzufügen und Entfernen von *VectorSets* und *Vector* Objekten belegt. Das TreeViewer-Widget visualisiert die erzeugten Objekte. Die dort angezeigte Baumstruktur gliedert sich in Wurzelelemente, die jeweils die VectorSets repräsentieren. Als Astelemente der Sets werden die Vector-Objekte angezeigt.

Zur erleichterten Bedienung wurden die Kommandos zum Erstellen und Entfernen von Vektoren initial deaktiviert. Erst bei entsprechender Auswahl eines Elements der Baumstruktur werden die Kommandos aktiviert. So kann ein Element erst entfernt werden, wenn eine Auswahl im TreeViewer erfolgte. Des Weiteren kann ein Vektor erst erstellt werden, wenn ein VectorSet ausgewählt ist, das den neuen Vektor beinhalten soll. Dafür wurden die SourceProvider (siehe Abschnitt 4.2.4) verwendet.

Für die Erzeugung eines Vector-Objekts wurde ein Wizard-Dialog mit zwei Seiten erstellt. Auf der ersten Seite wird der Name des Vektors angegeben, unter dem er in der View angezeigt wird sowie die Koordinaten für den dreidimensionalen Raum. Optional kann der Wizard an dieser Stelle abgeschlossen werden, da die zweite Seite bereits als vollständig markiert ist. Auf der zweiten Seite können die zuvor gesetzten Koordinaten durch Vektoroperationen verändert werden (siehe Abschnitt 3.3.1). Dazu wurde im oberen des Dialogs die Eingabe der Operationen realisiert. Im unteren Bereich wurde eine Liste implementiert, in der alle anzuwendenden Operationen als String aufgelistet sind. Dabei entspricht das Format dem Aufbau *Operation Typ(Argument , {Argument})*. Die Additionsoperation mit einem Vektor hat beispielsweise den Aufbau *Add Vector(1.0, 1.0, 1.0)*. Die Multiplikation mit einem Scalar hat den Aufbau *Multiply Scalar(1.0)*. Die Operationen werden der Liste nach (von oben nach unten) auf die zuvor angegebenen Koordinaten angewendet und das Ergebnis angezeigt. Beim Entfernen oder Hinzufügen eines Listenelements, wird das Ergebnis neu berechnet. Für das Deserialisieren der

Operationen wurde der *java.util.regex.Matcher* verwendet, um die Strings mithilfe von Regular Expressions in die einzelnen Segmente zu zerlegen. Abschließend erstellt der Dialog mit den resultierenden Koordinaten ein Objekt vom Typ Vector.

Die Klasse Vector wurde implementiert, um in der kompletten Anwendung die Berechnungen im dreidimensionalen Raum zu vereinfachen. Damit der Overhead bei vielen Vektorberechnungen durch die Objekterzeugung minimiert wird, existiert zu jeder Operation die gleiche als sogenannte lokale Operation. Dabei wird bei der Addition zweier Vektoren der Vektor modifiziert, der die Operation bereitstellt und nicht wie bei der anderen Variante ein neues Objekt erzeugt.

Environment-Explorer Die View dient zur Abbildung eines zu simulierenden Systems in ein 3D-Modell. Über das Kommando *Add Envrionment* wird dem Model-Plug-In ein neues Envrionment-Objekt hinzugefügt. Dazu wird in dem zugehörigen Command-Handler ein neues Environment-Objekt erzeugt und dem DatenHandler hinzugefügt. In der View wird in einem TreeViewer-Widget ein Element erzeugt, das als Attribut eine Referenz des Environments erhält. So kann über das Viewer-Element direkt auf die Environment-Instanz zugegriffen werden. Gruppenobjekte können nur erstellt werden, wenn in dem Viewer ein Environment-Element selektiert ist. Dies wird über den *SourceProvider* (siehe Abschnitt 4.2.4) realisiert. Der Command-Handler greift auf das ausgewählte TreeViewer-Element zu. Dies erfolgt über den *Selektion Service* (siehe Abschnitt 4.2.3). Über die Referenz kann nun auf das Environment-Objekt zugegriffen werden, um ein neues Gruppenobjekt hinzuzufügen. Gleiches Vorgehen gilt auch für das Hinzufügen von Atom-Objekten über die View.

Für das Erzeugen von Gitterstrukturen ist es erforderlich, dass in dem TreeViewer ein Gruppenelement selektiert wurde. Mit dem Kommando *Add Lattice* wird ein Wizard-Dialog geöffnet, über den die Gitterstruktur eingestellt wird. Die Erzeugung eines Gitters erfolgt nach dem Prinzip der Bravais-Gitter (siehe Abschnitt 3.3.1). Dafür werden die Vector-Objekte benötigt, die zur Beschreibung der Gitterstruktur (siehe Abschnitt 2.2.2) dienen sowie eine Abgrenzung im dreidimensionalen Raum, die die Gittergröße angibt. Zum Erzeugen der einzelnen Gitteratome wurde ein rekursiver Algorithmus umgesetzt. Als Abbruchkriterium wird überprüft, ob die zu setzenden Position sich innerhalb der vorgegebenen Grenzen befindet und noch kein Atom dort positioniert ist. Es werden alle positiven und negativen Vielfachen der Basisvektoren solange durchlaufen, bis keine freie Position mehr vorhanden ist.

```
1 private static void createLattice(
2   IGroup group, double[] size, IVector position, IVector... vectors )
3 {
4   // Check dimension size to create only atoms inside borders.
```

```
5      if( position.getX() > size[0] || position.getX() < 0 ||
6          position.getY() > size[1] || position.getY() < 0 ||
7          position.getZ() > size[2] || position.getZ() < 0 ){ return; }
8
9      // Check position for collisions with previous atoms.
10     boolean exist = false;
11
12     for( IAtom atom : group.getAtoms() )
13     { if( atom.getPosition().equals( position ) ){ exist = true; break; } }
14
15     // Create new atom if no other exist at this position.
16     if( exist ){ return; }
17     else{ group.addAtom().setPosition( position ); }
18
19     // Go to next recursion.
20     for( IVector vec : vectors )
21     {
22       createLattice( group, size, position.add( vec.mul( -1 ) ), vectors );
23       createLattice( group, size, position.add( vec           ), vectors );
24
25     }
26   }
```

Listing 4.19: Algorithmus zur Gittererzeugung nach Bravais

4.3.3 Simulations

Für die Beschreibung der zeitlichen Veränderung eines Systems wurde eine Perspektive realisiert, die alle dafür notwendigen Funktionen bereitstellt. Die Perspektive unterteilt sich in die drei Views Simulation-Explorer, Simulation-Properties und Parameter-Explorer.

Simulation Explorer Wie bei der Environment-Explorer-View (siehe Abschnitt 4.3.2) werden die Simulationselemente ebenfalls in einem TreeViewer-Widget dargestellt. Wurzelelement des TreeViewer ist das Simulationselement (Repräsentation in der View), das eine Referenz auf das Simulationsobjekt (Datenhaltung des Models) beinhaltet. Simulationsobjekt bildet den Container für alle Aktionen, die bei der Ausführung durchgeführt werden. Als Unterelemente des Simulationselements werden im TreeViewer die Aktionen dargestellt. Die einzelnen Elemente beinhalten eine Referenz auf das zugehörige Model Objekt. Bei Selektion der Elemente erfolgt eine Attributdarstellung der Objekte in der Simulation-Properties-View. Über

das DataBinding-Konzept (siehe Abschnitt 4.2.3) werden die dargestellten Attribute mit denen des Models verknüpft. So kann eine schnelle Änderung einzelner Werte erfolgen.

Event-Handling Der Nachrichtenaustausch der Aktionen kann auf synchronem, sowie asynchronem Weg erfolgen. Jedes Simulationsobjekt hat eine eigene Instanz eines EventHandlers (siehe Abbildung 4.5). Beim Hinzufügen einer Aktion in dem Simulationsobjekt wird die Referenz der Simulation übergeben. So können alle Aktionen auf den EventHandler zugreifen. Die Adressierung der Events erfolgt über einen EventKey, der jeder Event-Instanz übergeben wird. Sender und Empfänger müssen für eine erfolgreiche Kommunikation jeweils den identischen EventKey angeben. Auf Senderseite stellt das Interface *ISimulationAction* die Operation *fireEvent(IEvent)* bereit, die ein Event mit einem bestimmten EventKey an den EventHandler übergibt. Für das synchrone Übertragen von Events steht auf Empfängerseite die Operation *addRequiredEvent(IEvent)* und *onEvent(IEvent)* zur Verfügung. Beide Operationen werden vom Interface ISimulationAction bereitgestellt. Über die Operation addRequiredEvent wird die Aktion als Empfänger eines Events registriert, das dem entsprechenden EventKey enthält. Sobald ein Event mit dem Schlüssel eingetroffen ist, wird die Operation onEvent der Aktion aufgerufen und das Event übergeben. Für das asynchrone Übertragen von Events stellt der EventHandler einen Cache bereit. Zu jeweils einem EventKey wird das zuletzt eingetroffene Event gespeichert. Auf Empfängerseite kann über die Operation *getCachedEvent(IEvent)* ein Event aus dem Cache abgerufen werden. Hierbei wird als Argument ein Template übergeben, das den gleichen EventKey enthalten muss wie das zu empfangende Event.

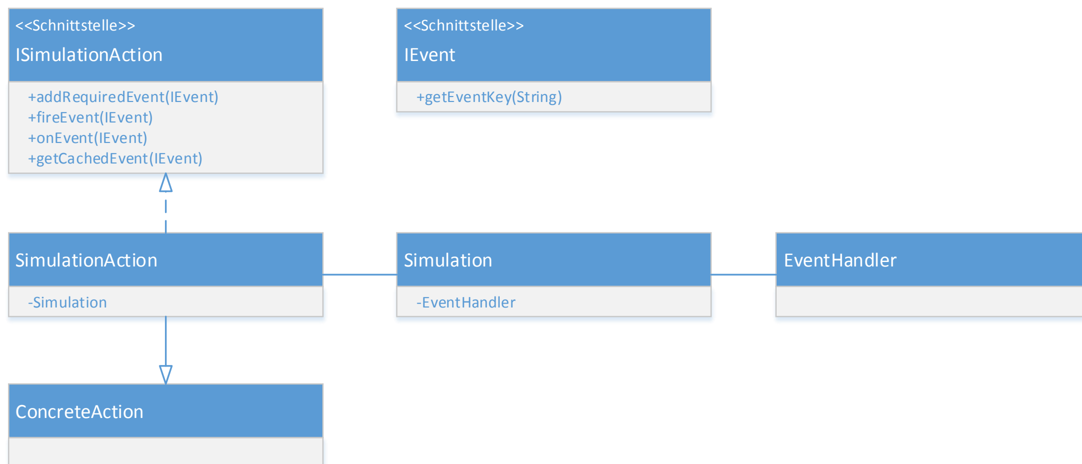

Abbildung 4.5: Event-basierte Kommunikation der Aktionen

Parameter Map und Id Handling in Simulationsactions Für die Relaxation von einzelnen Atomen werden Parameter benötigt, die zur Beschreibung der verwendeten Wechselwirkungen (siehe Abschnitt 2.2.3) dienen. Um auch Wechselwirkungen mit anderen Parametern zu unterstützen, wurden sogenannte *MaskParameterSets* verwendet. Dadurch bleibt die Anwendung für spätere Entwicklungen erweiterbar und flexibel. Für jede Aktion, die mit dem Konzept der ParameterSets (siehe Abschnitt 3.3.2) arbeitet, muss ein Attribut der Klasse *MaskParameterSet* implementiert werden. Hier wird der Aufbau beschrieben, den die Aktion von den ParameterSets erwartet. Wie im folgenden Code-Abschnitt zusehen ist, werden die zwei Wechselwirkungen *Morse* und *Exchange* verwendet, die jeweils drei Parameter benötigen.

```
public static final MaskParameterSet PARAMETERMASK = new MaskParameterSet(
            new MaskCategory( MORSE,
                    new MaskParameter( U0 ),
                    new MaskParameter( ALPHA ),
                    new MaskParameter( R0 ) ),
            new MaskCategory( EXCHANGE,
                    new MaskParameter( J0 ),
                    new MaskParameter( ALPHA ),
                    new MaskParameter( R0 ) ) );
```

Listing 4.20: MaskParameterSet für die Zuweisung von Parametern zur Laufzeit

In der View Komponente der Aktion werden die ParameterSets über ein Dialogfenster zugeordnet. Dabei wurde eine *SelectionRestriction* (siehe Abschnitt 4.2.4) verwendet, um das ausgewählte ParameterSet des Dialogfensters mit den Anforderungen der entsprechenden Aktion zu prüfen. So können nur ParameterSets ausgewählt werden, die den Anforderungen genügen.

```
dialog.setSelectionRestriction( new IDialogSelectionRestriction()
{
  @Override
  public boolean restriction( List<Object> selection )
  {
    if( selection.size() != 1 ){ return false; }

    if( selection.get( 0 ) instanceof IParameterSet )
    {
      IParameterSet parameterSet = ( IParameterSet ) selection.get( 0 );

      return parameterSet.match( Relaxation.PARAMETERMASK );
    }
    return false;
```

```
15    }
16 } );
```

Listing 4.21: Auswahlbeschränkung von ParameterSets über die MaskParameterSet

4.3.4 Distributor

Für die Simulationsausführung wurde eine Perspektive *Distributor* implementiert. Die Perspektive wurde in zwei Views unterteilt: den Distributor-Explorer und den Simulation-Explorer. Der Simulation-Explorer wurde aus dem Simulation-Plug-In mithilfe des Extension-Point-Mechanismus an die Distributor-Perspektive angehängt. Der Distributor-Explorer ist ähnlich aufgebaut wie der Simulation-Explorer. Hier wird in einem TreeViewer-Widget die Ausführungskomponente erstellt. Für die Zuordnung einer Ausführung mit einer Simulation werden beide Elemente in den zwei Views selektiert. Über einen Kommando *Connect* kann nun das Distributor Objekt mit dem Simulationsobjekt verbunden werden. So wird dem Distributor die Instanz der Simulation übergeben.

Distributor Setup Der Start einer Simulation erfolgt über das Distributor-Objekt. Für jede auszuführende Simulationsinstanz wird eine Kopie der Original-Simulation angefertigt. Jede kopierte Simulationsinstanz erhält eine Kopie des Environments von der Original-Simulation. Anschließend werden die Distributor-Aktionen ausgeführt und auf die jeweiligen Simulationskopien angewendet. Die realisierte Aktion für den parallelen Linien-Scan greift beispielsweise über die Simulationsinstanz auf ihr kopiertes Environment zu und ändert die Position der Spitzenatome sowie des Adatom. Anschließend werden die Simulationen jeweils an Execute-Objekte übergeben. Diese bilden den Rahmen der Ausführung und implementieren das Interface *Callable*. Somit kann ein Execute direkt in einem Thread ausgeführt werden. Die einzelnen Executes werden an den Executor übergeben, der die Ausführung der Simulationen übernimmt. Bei dem implementierten LocalExecutor wird ein ThreadPool angelegt, der nach der Anzahl der lokalen Prozessorkerne skaliert. Die einzelnen Execute-Objekte werden dort direkt an den ThreadPool übergeben und ausgeführt.

Simulations-Scheduling Die Ausführung einer Simulation erfolgt im Execute Objekt. Dabei werden die Operation *start()* und *step()* des Interfaces *ISimulation* verwendet. Beim Starten der Simulation über die Operation *start()* wird ein *ActionScheduler* erzeugt, der für die Reihenfolge der Aktionsausführung verantwortlich ist. Dieser bekommt alle Simulationsaktionen übergeben und verwaltet alle Aktionen in einem Array. In einem zweiten Array sind für

jeden Schritt die Indizes des ersten Arrays hinterlegt, geordnet nach Ausführungsschritt und Priorität. Dieses Array wird LookUpArray genannt. Wenn z.B. drei Aktionen $A\{s,p\}$ hinzugefügt werden (s für Schritte und p für Priorität): $A1\{1,1\}$, $A2\{200,2\}$, $A3\{400,1\}$, so enthalten das Array für jeden Schritt bei dem eine Aktion ausgeführt wird, den Index der Aktion: $\{1,1,1,...,1,2,1,1,1...,1,3,2\}$. Die Länge des LookUpArrays richtet sich nach der kleinsten wiederholenden Periode der Ausführungsschritte. So sind in diesem Beispiel 403 Einträge für die drei Aktionen eingetragen: 400xA1, 2xA2 und 1xA3. Das Array wird bei Aufruf der Operation step nach und nach zyklisch durchlaufen und die zugehörige Aktion ausgeführt. Dies wird solange fortgesetzt bis die maximale Anzahl an Simulationsschritten erreicht ist.

4.3.5 Monitor

Für die Monitor-Komponente wurde eine Perspektive *Monitor* realisiert, die sich in sechs Views unterteilt: Environment-Explorer-, Environment-Properties- (aus dem Environment-View-Plug-In), ParameterSet-Explorer- (aus dem Simulation-View-Plug-In), Canvas-Container- (für die grafische Darstellung), Running-Environments- und EnergySettings-View.

Running Environments Für die Übersicht über laufende Simulationen werden in der View mit einem TreeViewer Widget die verschiedenen Environments angezeigt, die beim Starten eines Distributors erstellt wurden (siehe Abschnitt 4.3.4).

Canvas Container Zur Visualisierung der Objekte des Environments wurde die JMonkeyEngine verwendet (siehe Abschnitt 4.1). Die Engine stellt einen Abstract-Window-Toolkit (AWT)-Canvas-Container bereit. Für die Integration in ein RCP-Viewpart wurde von SWT eine Klasse SWT_AWT bereitgestellt. Dadurch wird es ermöglicht, AWT-Komponenten in SWT zu integrieren. Der *Monitor* stellt die Kernklasse der Visualisierung dar. Die Klasse liefert die Operation *getCanvasContainer()* mit der die AWT-Komponente erstellt wird. Die Operation *simpleUpdate* wird direkt von der JMonkeyEngine aufgerufen und stellt die Hauptschleife der Visualisierung dar. Hier werden alle Aktualisierungen durchgeführt, die eine Änderung der Darstellung bewirken. Um eine Schnittstelle zu liefern, wurden zwei Operationen bereitgestellt, die eine Erweiterung der simpleUpdate-Operation ermöglichen. Mit der Operation *addInitialExecutes(IGraphicEngineExecute)* ist es möglich, Erweiterungen auszuführen, die genau einmal im ersten Durchlauf der simpleUpdate-Operation ausgeführt werden. Die Operation *addUpdateExecutes(IGraphicEngineExecute)* ermöglicht das Hinzufügen von Erweiterungen, die bei jedem Durchlauf ausgeführt werden. Für die Erweiterungen wurde eine abstrakte Klasse implementiert, die in den abgeleiteten Klassen die Funktionalität der GraphicEngine zur Verfü-

gung stellt. Dies wird dadurch realisiert, indem nach dem Hinzufügen einer Execute-Klasse alle wichtigen Handler von der JMonkeyEngine übergeben werden. Für eine übersichtlichere Darstellung des Environments wurde eine Erweiterung implementiert, die den Canvas Container in vier Segmente unterteilt. In den oberen zwei Segmenten wird das angezeigte Environment entlang der Z- und X-Achse abgebildet. Im linken, unteren Segment wird das Environment über die Y-Achse abgebildet. Im letzten Segment wird eine Sicht diagonal über alle Achsen auf das Environment abgebildet. Eine weitere Erweiterung realisiert die Verschiebung der Kamera-Perspektiven in den vier Segmenten über die Cursortasten. So kann das betrachtete Environment aus unterschiedlichen Positionen und Abständen betrachtet werden.

EnergySettings Um die Parameter der unterschiedlichen Wechselwirkungen anzuzeigen, wurde eine Methode zur Visualisierung von Energien des Systems implementiert. Über die EnergySettings-View lässt sich eine Energielandschaft von mehreren Atomen darstellen. Über die View kann eingestellt werden, welche Atome betrachtet werden sollen, welche Parameter-Sets für die unterschiedlichen Element-Typen der Atome verwendet werden, welcher Bereich berechnet werden soll und mit welcher Auflösung diese dargestellt wird. Dafür wurden zwei Möglichkeiten realisiert: Die 3D-Visualisierung erstellt eine Energielandschaft entlang der Oberfläche. Dabei bildet sich die Langschaft aus der Energie eines Atoms, das in Ruhelage relaxiert wurde. Die X und Y Werte der Langschaft sind dabei die Position des Atoms. Die Z Werte ergeben sich aus der Energie des Atoms bei einer Position. Die Höhenwerte der Landschaft werden mit einem Farbverlauf eingefärbt. Die 2D Visualisierung erstellt einen Spline zwischen einem Start- und Ziel-Vektor im Canvas-Container. Ebenfalls wird die Ruhelage eines darauf liegenden Atoms entlang der Spline gerastert. Die Energie Werte werden in der EnergySettings-View in einem Liniendiagramm dargestellt. Für das Diagramm wurde die Bibliothek JFreeChart verwendet(siehe Abschnitt 4.1).

4.3.6 Analysis

Für die Auswertung der Ergebnisse wurde eine Perspektive realisiert, die sich in drei Views untergliedert: der Measurements-Explorer, Measurements-Image und Measurements-Chart. Im Measurements Explorer werden alle Messergebnisse angezeigt, die sich in dem voreingestellten Ordner befinden. Diese werden ebenfalls in einem TreeViewer Widget angezeigt. Die Messergebnisse werden von der Measurements Aktion der einzelnen Simulationen erzeugt. So wird für jede Simulationsinstanz eine Datei mit der Endung *.asmp* erzeugt, die jeweils die Messergebnisse einer Scan-Linie beinhaltet. Die Benennung der Datei erfolgt nach dem Schema Name-Sequenz-Zeitstempel. Den Namen liefert die Simulation, die mit dem Distributor verbun-

denen ist. Für die Sequenz wird die Nummer der ausgeführten Simulationskopie verwendet. Der Zeitstempel ergibt sich durch die Zeit zwischen Distributor-Start und dem 1. Januar 1970 (UTC) in Millisekunden. Dadurch lassen sich alle zugehörigen Ergebnisse einer Distributor Ausführung zuordnen.

Ergebnis Präsentation Beim Selektieren eines Elements aus dem Explorer werden die Messwerte in der Measurements-Chart-View als Liniendiagramm dargestellt. In der View Measurements-Image werden alle zugehörigen Ergebnisse eines Distributors geordnet nach der Sequenz Nummer als Image dargestellt. Die Intensität der Messwerte wird in einem Farbverlauf skaliert. So bekommt der niedrigste Messwert den Farbton schwarz und der höchste Wert den Farbton weiß. Die dazwischen liegenden Werte werden mit orange Werten skaliert. Diese Methode der Farbtonskalierung wurde aus Vorarbeiten[4] übernommen und wird in dieser Arbeit nicht weiter thematisiert.

Export Measurement Die zusammengesetzten Messergebnisse einer Simulationsdurchführung ergeben vergleichbare Werte, die mit aktuellen Rastertunnelmikroskopen und entsprechender Steuerungssoftware erzeugt werden können. Zur besseren Analyse und Gegenüberstellung der Ergebnisse wird eine Exportfunktion realisiert. Dadurch ist es möglich, die Simulationsergebnisse in den selben Programmen zu untersuchen wie die Ergebnisse der mikroskopischen Versuchsdurchführung. Dafür werden die Werte in ein *XYZ Datei Format*[5] überführt, wobei XY die Koordinaten der Messpunkte und Z die Amplitudenstärke darstellt (siehe Abschnitt 2.2.1).

[4]Quelle: Wolter (2014)
[5]http://gwyddion.net/documentation/user-guide-en/gxyzf.html

5 Test und Validierung

5.1 Testen der Simulation

Kern der Simulation bildet die *ActionScheduler*-Klasse (siehe Abschnitt 4.3.4). Sie ist für die Ausführung der Aktionen in vordefinierter Reihenfolge zuständig und bildet dadurch das Verhalten des Simulationspozesses auf dem Environment ab. Zur Sicherstellung der korrekten Ausführung der Aktionen werden die Grenzwerte der Eingabeparameter analysiert. Für die Ausführung der Simulation werden die Operationen *addAction(IAction)*, *start()* und *step()* benötigt. Die Aufrufreihenfolge ist klar strukturiert (addAction -> start -> step). Da die Klasse Simulation diese Operationen in der vorgegebenen Reihenfolge aufruft, wird eine Permutation der Aufrufreihenfolge nicht weiter untersucht. Somit bleibt als testbares Argument lediglich die Action-Klasse. Die ActionScheduler-Klasse greift bei der Ausführung auf zwei Attribute der Action-Klasse zu: Step und Priority. Durch die Bildung von Äquivalenzklassen lassen sich die Tests in zwei Wertebereiche untergliedern: zulässige sowie unzulässige Werte. Für die Attribute gelten: $Integer.MIN_VALUE$ bis 0 als unzulässig und 1 bis $Integer.MAX_VALUE$ als zulässig. Für diese beiden Äquivalenzklassen wird der Positivtest sowie Negativtest der Black-Box-Test-Strategie verwendet. Der Positivtest dient dazu zu zeigen, dass die Software bei korrekter Eingabe der Parameter korrekt funktioniert. Der Negativtest hingegen beschäftigt sich mit der Fehlerbehandlung bei falschen Eingaben (vgl. Wetzlmaier (2007)). Der ActionScheduler verfügt über keinerlei Fehlererkennung und würde auch mit falschen Eingabewerten arbeiten. Um dies zu verhindern wurden unzulässige Werte bei der Eingabe in der View Komponenten abgefangen. Über den *WizardStateHandler* (siehe Abschnitt 4.2.4) wurden Fehlerkonditionen implementiert, die negative Eingaben nicht akzeptieren. Das Testen der unzulässige Werte wurde durch manuelle Bedienung der Anwendung überprüft.

Das Testen der zulässigen Werte erfolgt über Unit-Tests nach dem Black-Box Prinzip. Das bedeutet, dass die Klasse nur von außen getestet wird. Damit eine Bewertung der Funktionalität erfolgen kann, wird eine messbare Ausgangsgröße vom ActionScheduler benötigt. Dafür wurde eine Klasse (TestAction) implementiert, die das Interface *IAction* realisiert. Von dem ActionScheduler werden die Operationen *getStep()*, *getPriority()* und *run()* des IAction Interfaces

verwendet. Für eine messbare Ausgangsgröße wird jeder TestAktion Instanz eine Id hinzugefügt. Beim Aufruf der run-Operation wird die Id der Aktion in einer Liste (ConcreteResult) des TestCase hinterlegt. Für jeden einzelnen TestCase wird eine Liste mit erwarteten Ergebnisse (ExpectedResult) manuell angelegt. Die beiden Listen werden nach Testfall-Ausführung miteinander verglichen. Die erstellte TestSuite beinhaltet drei TestUnits: Priority, Steps und PrioritySteps. In der TestUnit Priority wurden Aktionen mit unterschiedlichen Prioritäten getestet. In der Unit Steps wurden unterschiedliche Ausführungsschritte getestet. In der Unit PrioritySteps wurden verschiedene Permutationen der Parameter getestet. Bei allen TestUnits wurden folgende Test implementiert:

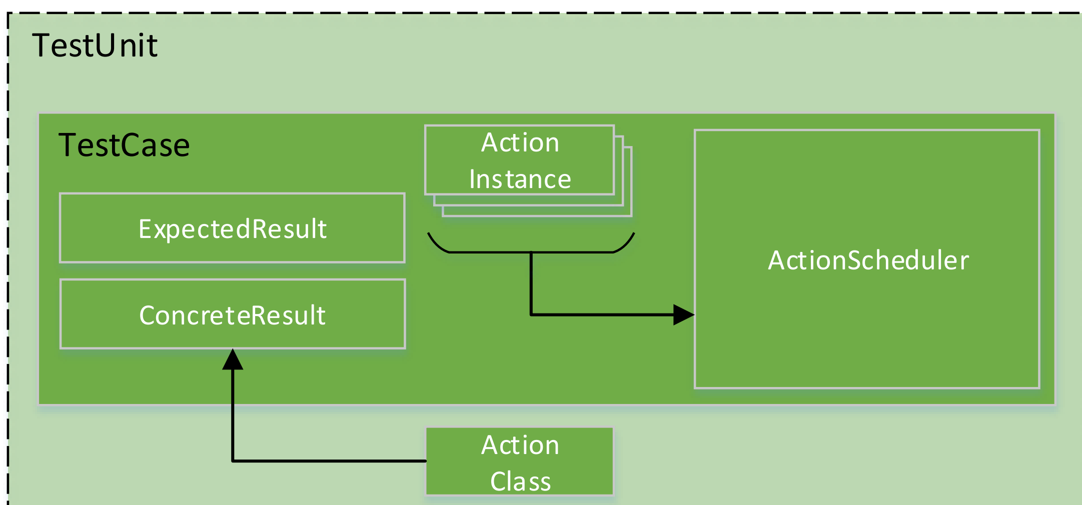

Abbildung 5.1: Aufbau einer TestUnit für den Black-Box Test des ActionSchedulers

1. TestSameOrderOneAction()
    - *Drei* Aktionen hinzugefügt
    - Jede Aktion mit *anderer* Parameterbelegung
    - Nach Parameter *aufsteigend* sortiert hinzugefügt
2. TestDifferentOrderOneAction()
    - *Drei* Aktionen hinzugefügt
    - Jede Aktion mit *anderer* Parameterbelegung
    - *Unsortiert* hinzugefügt

3. TestSameOrderTwoActions()
    - *Sechs* Aktionen hinzugefügt
    - Jeweils zwei Aktionen mit *gleicher* Parameterbelegung
    - Nach Parameter *aufsteigend* sortiert hinzugefügt
4. TestDifferentOrderTwoActions()
    - *Sechs* Aktionen hinzugefügt
    - Jeweils zwei Aktionen mit *gleicher* Parameterbelegung
    - *Unsortiert* hinzugefügt

Die TestAction Instanzen wurden mit Ausführungsschritten von 1 bis 3 belegt. Die Testdurchführung erfolgte mit 6 *step()*-Aufrufen. So wurde jede Aktion mindestens zweimal ausgeführt.

5.2 Verifikation des Simulationsprozesses

Vorgehen Um zu zeigen, dass die Anwendung korrekt implementiert wurde, wird der gesamte Workflow der Simulationsstudie anhand von veröffentlichten Simulationsergebnissen verifiziert (vgl. Wolter u. a. (2012)). Zu Beginn wird eine exakte Kopie der Probe modelliert. Anschließend werden die Systemparameter eingestellt, die das dynamische Verhalten des Prozesses abbilden. Die gewonnenen Ergebnisse der Simulationsdurchführung werden mit den vorliegenden Ergebnissen verglichen und ausgewertet. Dadurch soll gezeigt werden, dass der Arbeitsablauf der Simulationsstudie auch für andere Experimente einsetzbar.

Das Experiment Simuliert wird das Experiment Mn/W(110), das zuvor mit einem RTM untersucht und mit einer Simulation reproduziert wurde. Durchgeführt wurde das Experiment mit einer Temperatur von 8K (Kelvin). Als Probe wurde eine Gitterlage Mangan (Mn) auf einer Lage Wolfram (W) untersucht. Die Kristallflächen der beiden Lagen werden durch die sogenannten *Millersche Indizes* (abgekürzt mit h, k und l) angegeben. Wolfram bildet ein kubisch raumzentriertes Gitter (siehe Abschnitt 2.2.2). Über die Indizes 110 werden die Punkte der Kristallebene auf den XYZ-Achsen angegeben. Dadurch ergibt sich ein Schnitt diagonal durch das Gitter (siehe Abbildung 5.2) in XY Richtung (vgl. Kittel (2006)). Die Gitterkonstante von Wolfram beträgt 3,165 Å (316,5 pm). Damit ist der kleinste Abstand zwischen zwei Atomen des Gitters gemeint. Somit ergeben sich die beiden Basisvektoren für das Gitter $\vec{v1} = 3,156$ und $\vec{v2} = 2,741$ der Ebene 110. Die Mangan-Lage bildet auf der Wolfram Struktur eine Spin-Spirale mit 170° zwischen den benachbarten Reihen. Dadurch ergibt sich eine 360° Drehung der

magnetischen Ausrichtung entlang der X-Achse (durch Millersche Indizes mit $[1\bar{1}0]$ angegeben). Über die beiden Atomlagen Mn/W(110) entlang wird ein Adatom manipuliert. Dieses Adatom befindet sich zwischen der Mangan Lage und der Spitze. Die Spitze wird vereinfacht durch ein Atom dargestellt. Auf das Adatom wirken die Kräfte der beiden Atomlagen und der Spitze. Diese Kräfte ergeben sich aus der magnetischen sowie chemischen Bindung der Atome. Die beiden Wechselwirkungen werden durch die Parameter aus Tabelle 5.1 beschrieben. Die Spitze

		Wolfram	Mangan	Spitze
Chemische Bindung	D	0.4	0.4	1.55
	R0	3.1	3.1	2.4
	Alpha	1.5	1.5	0.8
Magnetische Bindung	J0	0	0.188	0.188
	R0	0	2.875	2.4
	Alpha	0	2	2

Tabelle 5.1: Parameter der Wechselwirkungen des Systems

bewegt sich mit einem Abstand von 5.45 Å zur oberen Atomlage entlang der Oberfläche. Positioniert wurde die Spitze genau zwischen zwei Atomreihen (siehe Abbildung 5.3).

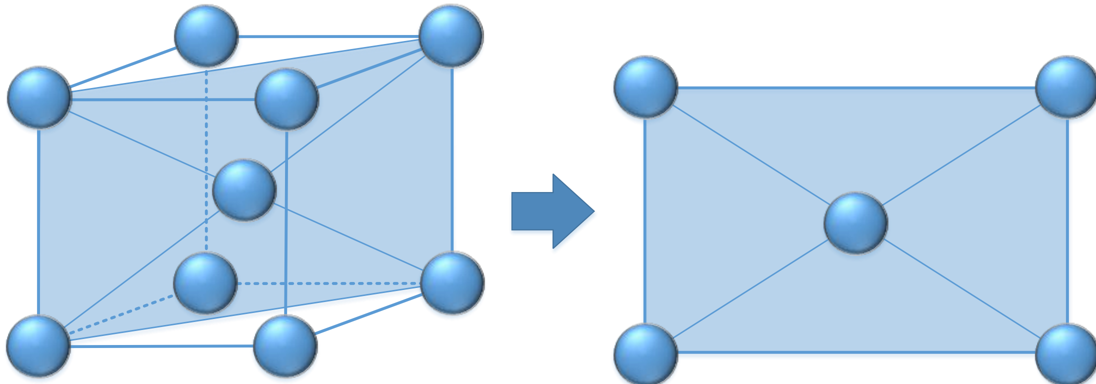

Abbildung 5.2: Schnitt eines BCC Gitters durch die Ebene (110)

Vorhandenen Daten Für die Validierung standen die Energieaufzeichnungen des zu manipulierenden Atoms entlang der Oberfläche zur Verfügung. Dabei wurde die Energie des zu manipulierenden Atoms nach jeder Positionsänderung der Spitze entnommen und über die Spitzenbewegung in X-Richtung abgebildet.

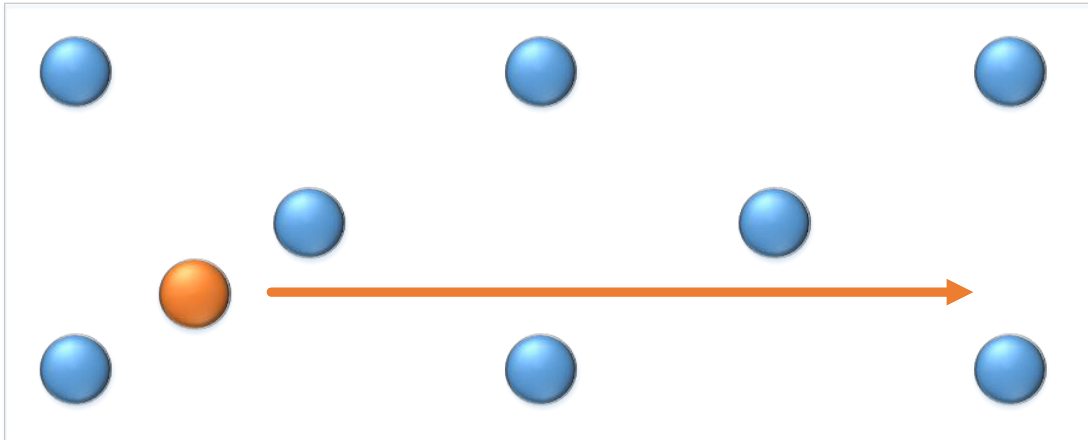

Abbildung 5.3: Bewegungspfad der Spitze über die Mn Lage

Modellierung Über den Vector-Explorer wurden die zwei Basisvektoren $\vec{v1} = 3,156$ und $\vec{v2} = 2,741$ angelegt. Mit dem Environment-Explorer wurden über die Gitter-Funktion zwei Atomlagen angelegt. Eine der beiden Lagen wurde in Y-Richtung um 3.165/2 Å und Z-Richtung um -2.238 Å verschoben, um zwei Schichten des kubisch raumzentrierten Gitters zu konstruieren. Für die Spin-Spirale wurde vereinfacht angenommen, dass die Magnetisierung zweier benachbarten Reihen antiparallel zueinander ausgerichtet sind (siehe Abbildung 5.4). Anschließend wurden zwei Atome für Spitze und Adatom erstellt. Allen Atomen wurde ein entsprechender Element Typ zugeordnet, damit bei Simulationsdurchführung die richtigen Wechselwirkungen angewendet werden. Nach der Modellierung der Probe, wurde das Verhalten des Prozesses abgebildet. Dafür wurde ein Simulationsobjekt mit drei Aktionen erstellt: eine Relaxation-Aktion, die das Adatom mit der Oberfläche und der Spitze relaxiert, eine Translation-Aktion, die das Spitzenatom bei jeder Ausführung um 0,01 Å in X Richtung bewegt und eine Measurement-Aktion, die nach n Schritten der Spitze die aktuelle Energie des Adatoms aufzeichnet. Anschließend wurde ein Distributor erstellt, der nur eine Simulationsinstanz ausführt, um genau eine Scan-Linie zu untersuchen. Die erzeugten Ergebnisse wurden über das Diagramm der Analysis-Perspektive betrachtet und mit den vorhandenen Werten der vorherigen Simulation verglichen.

Ergebnisse und Übereinstimmung In Abbildung 5.5 werden die Ergebnisse der Simulation (oberes Diagramm rote Messpunkte) mit den Ergebnissen eines anderen Simulationsprogramms (oberes Diagramm grüne Messpunkte) verglichen (vgl. Wolter u. a. (2012)). Dargestellt ist die Energie des Adatoms über die X-Position des Spitzenatoms. Zu sehen ist der Energieunterschied

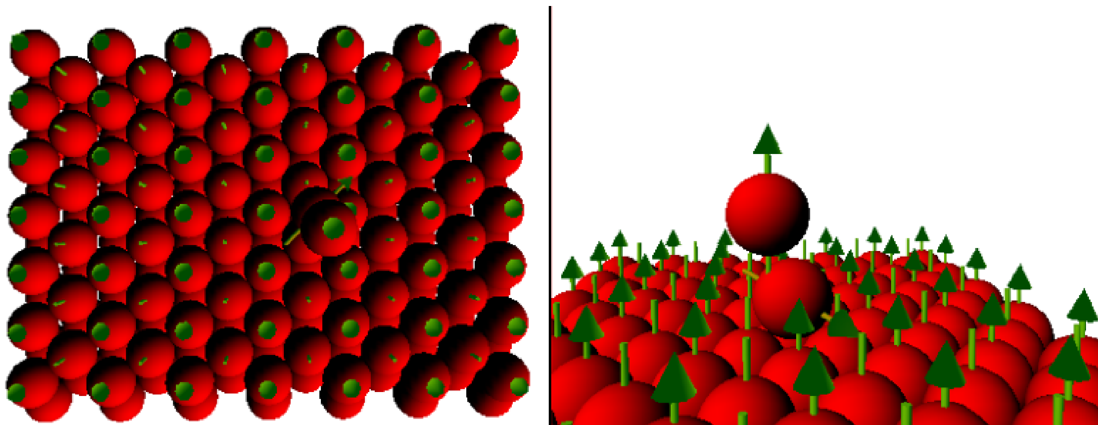

Abbildung 5.4: Antiparallele Ausrichtung der benachbarten Mn Reihen

bei Sprüngen des Adatoms in eine benachbarte Reihe. Die höhere Energie (um -3,35 meV) ergibt sich bei einem Sprung von einer parallel zur Spitze ausgerichteten Reihe in eine antiparallel ausgerichtete Reihe. Weitaus günstiger ist ein Sprung in eine parallel ausgerichtete Reihe (um -3,45 meV). Die Abweichungen der beiden Ergebnisse resultieren aus dem zufallsbasierten Ansatz der Monte-Carlo-Simulation (siehe Abschnitt 3.2.3). Da nicht mit dem selben Startwert für die Zufallsgeneratoren gearbeitet wurde, ergibt sich ein unterschiedlicher wahrscheinlichkeitsgewichteter Pfad durch den Zustandsraum für das Adatom. Die Unterschiede in Richtung der Ordinate ergeben sich durch die Anzahl der Nachbarn, die für die Energieberechnung des Adatoms einbezogen wurden. Wichtig war es zu zeigen, dass die magnetische Ausrichtung der benachbarten Reihen in der Energie des Adatoms zu sehen ist und die Abweichungen der beiden Ergebnisse (unteres Diagramm) im Mittel weniger als 1% aufweisen.

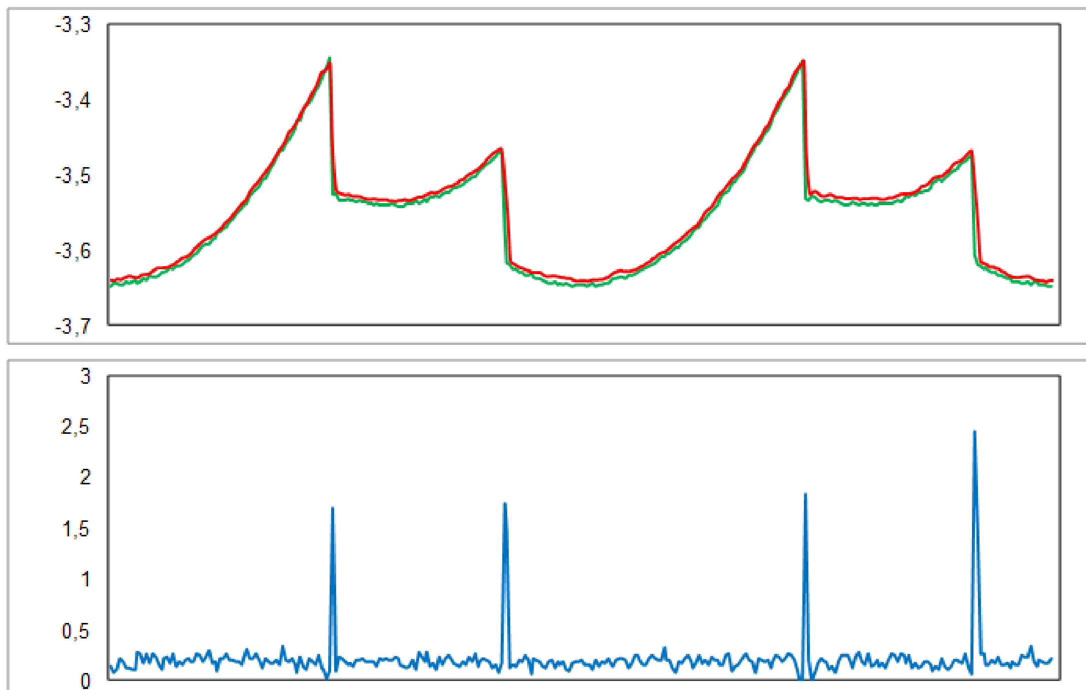

Abbildung 5.5: Energie des Adatoms (rot) verglichen mit Ergebnissen von Wolter u. a. (2012) (grün) und der Abweichung in Prozent (blau)

6 Zusammenfassung und Fazit

Ziel dieser Arbeit war es, eine Entwicklungsumgebung zur Durchführung von Simulationsstudien zu entwickeln, die einen übersichtlichen und strukturierten Arbeitsablauf ermöglicht. Dieses abschließende Kapitel fasst die Ergebnisse dieser Arbeit zusammen (Abschnitt 6.1) und gibt einen Ausblick auf weiterführende Entwicklungen (Abschnitt 6.2).

6.1 Zusammenfassung der Zeile und Ergebnisse

Die Entwicklungsumgebung zur Durchführung von Simulationsstudien wurde so konzipiert, dass ein übersichtlicher Arbeitsablauf ermöglicht wird. Da die Forschung an atomaren Strukturen unterschiedliche Anforderungen erfüllen muss, wurde das Konzept so entwickelt, dass eine Erweiterung der einzelnen Teile des Arbeitsablaufs jederzeit gegeben ist, ohne bestehende Komponenten verändern zu müssen. Darüber hinaus wurde die Entwicklungsumgebung so konzipiert, dass eine Bedienung auch ohne Kenntnisse des zugrunde liegenden Sourcecodes erfolgen kann. Um diese Anforderungen zu erfüllen, wurde die Eclipse-Rich-Client-Platform verwendet. Sie bietet die Möglichkeit, die Entwicklungsumgebung adaptiv zu erweitern, in dem die bestehenden Plug-Ins durch weitere Plug-Ins erweitert werden können. Die Grundfunktionalitäten bieten fünf Plug-Ins, die wesentliche Punkte einer Simulationsstudie erfüllen. Zum Modellieren eines Systems wurde ein Plug-In erstellt, das es ermöglicht, den atomaren Aufbau einer experimentellen Probe strukturiert nachzubilden. Dafür stehen Funktionen zur Verfügung, die es ermöglichen unterschiedlichste Gitterstrukturen schnell und übersichtlich zu erstellen, was eine Fehlerreduzierung bewirken kann. Für die Modellierung des dynamischen Verhaltens eines Systems wurde ein weiteres Plug-In erstellt, mit welchem Prozesse unterschiedlichster Art formuliert werden können, die das Verhalten eines RTM simulieren sollen. Die Simulationsdurchführung erfolgt in einem weiteren Plug-In. Dieses bietet die Funktionalität, die Simulation zu parallelisieren und auf mehreren Prozessorkernen gleichzeitig auszuführen. Zur Analyse der produzierten Ergebnisse wurde ein weiteres Plug-In erstellt, mit dem die einzelnen Messwerte der Simulation betrachtet und exportiert werden können. Bei der Analyse können einzelne Messwerte einer Scan-Linie untersucht oder die Zusammensetzung aller Scan-Linien exportiert werden. Letzteres bietet die Möglichkeit, die produzierten

Daten mit vorhanden Daten eines RTMs in externen Werkzeugen zu vergleichen. Während der Simulationsstudie kann das modellierte System durch ein Plug-In visualisiert werden. Bei der Anpassung der Wechselwirkungen kann überprüft werden, wie sich die Parameter auf das System auswirken würden. In der Ausführungsphase kann das zu simulierende System und der durchgeführte Prozess live beobachtet werden, um so frühzeitig Entscheidungen über die Sinnhaftigkeit der kompletten Durchführung zu treffen. Darüber hinaus können auf diesem Wege auch produzierte Ergebnisse erneut simuliert werden, um das Verhalten des System besser verstehen zu können. Zur einfachen Bedienung dieser Plug-Ins wurde jeweils ein Plug-In für jede Phase der Studie entwickelt, welche einfache und konsistente Benutzeroberfläche bereitstellen. Diese kann auch Plug-In-übergreifend angepasst werden, sodass einzelne Teile der Benutzeroberfläche aus unterschiedlichen Phasen der Simulationsstudie zusammen angezeigt werden können.

Um die Entwicklungsumgebung zu testen, wurden Unit-Tests entwickelt, die eine korrekte Simulationsausführung überprüfen. Dafür wurde eine Reihe von Black-Box-Tests entwickelt, die nach dem Äquivalenzklassen System unterschiedliche Parameter-Permutationen testen. Für die Validierung der kompletten Entwicklungsumgebung und dessen Arbeitsablauf wurde ein Experiment nachgebildet, das bereits mit einer anderen Anwendung simuliert wurde.

6.2 Ausblick auf weitere Entwicklungen

Die Entwicklungsumgebung wurde so entwickelt, dass sie als virtuelles RTM fungieren kann. So sollen Ergebnisse eines Prozesses besser analysiert werden können, indem das dynamische Verhalten des Systems visualisiert werden kann. Damit die Anwendung dem wachsenden Einsatzgebiet eines RTMs mithalten kann, wird viel Spielraum für Erweiterungen aller Art geschaffen. So ist es möglich, weitere Werkzeuge zur besseren Modellierung der zu untersuchenden Probe zu realisieren. Für das dynamische Verhalten des Systems können weitere Wechselwirkungen eingesetzt werden, um atomare Strukturen besser verstehen zu können. Für andere Ausgangsgrößen der Simulation (wie beispielsweise der Tunnelstrom) können andere Messverfahren implementiert werden, um so neue Erkenntnisse aus der Forschung übernehmen zu können. Für die Visualisierung der Probe und des Prozesses könnten Erweiterungen realisiert werden, die den Pfad der zu manipulierenden Atome visualisieren oder die Scan-Bilder direkt auf der Probe anzeigen. Für die Ausführung der Simulation könnte auch ein Distributed-Executor implementiert werden, der die verteilte Ausführung auf mehreren Computern im Netzwerk realisiert und anschließend die Ergebnisse lokal verfügbar macht. So kann ein noch höherer Simulationsdurchsatz erzielt werden. Auch bei der Analyse der

Ergebnisse könnten Werkzeuge implementiert werden, die unterschiedliche Anforderungen von unterschiedlichen Experimenten realisieren.

Anhang A

Ausschnitte der Entwicklungumgebung

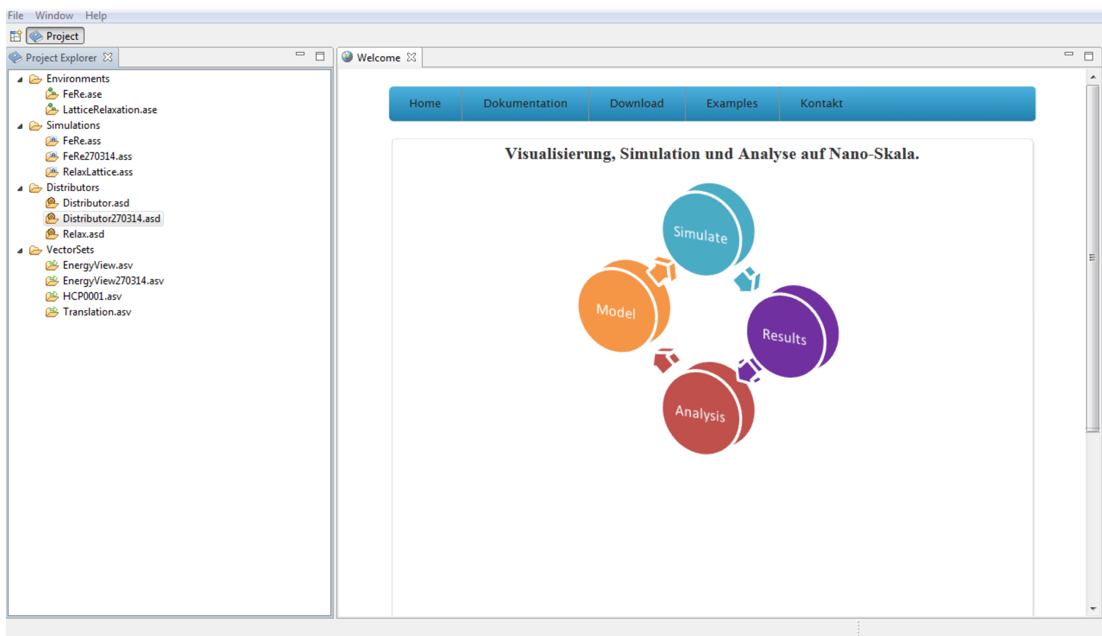

Abbildung A.1: Platform-Perspektive beinhaltet eine Übersicht aller erzeugten Datenobjekte und die Dokumentation der Software

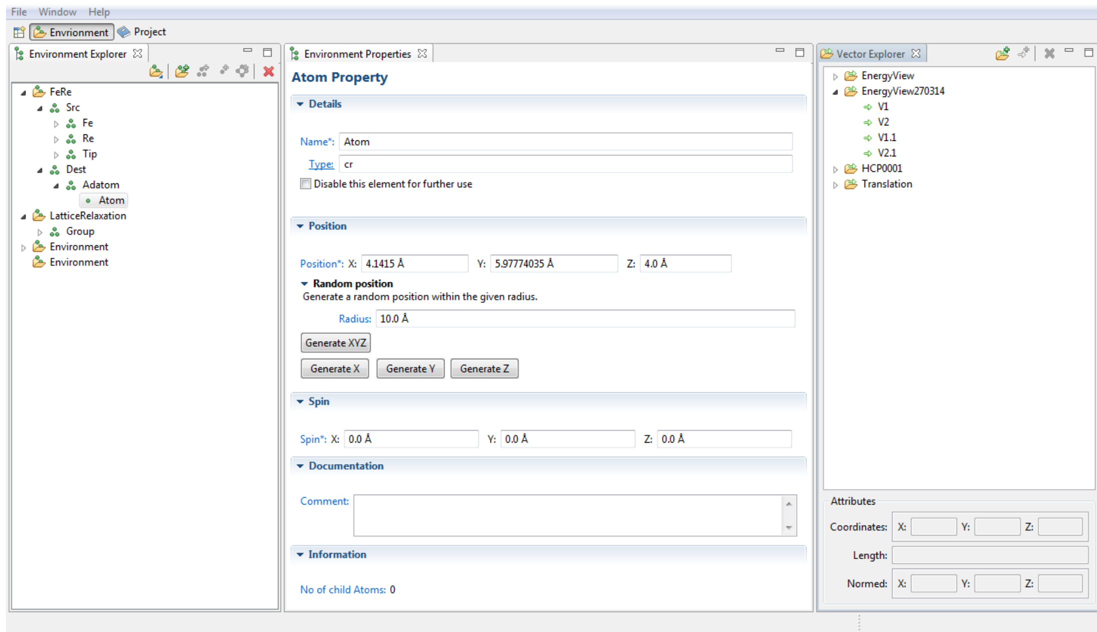

Abbildung A.2: Environment-Perspektive zur Modellierung des Systems

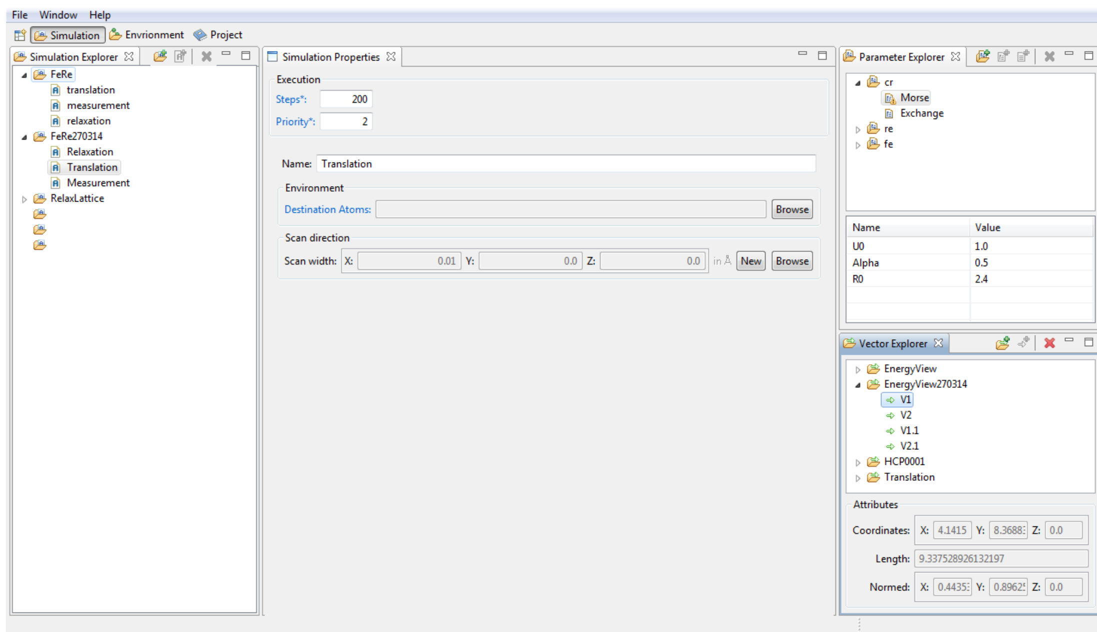

Abbildung A.3: Simulation-Perspektive zur Modellierung des Prozesses

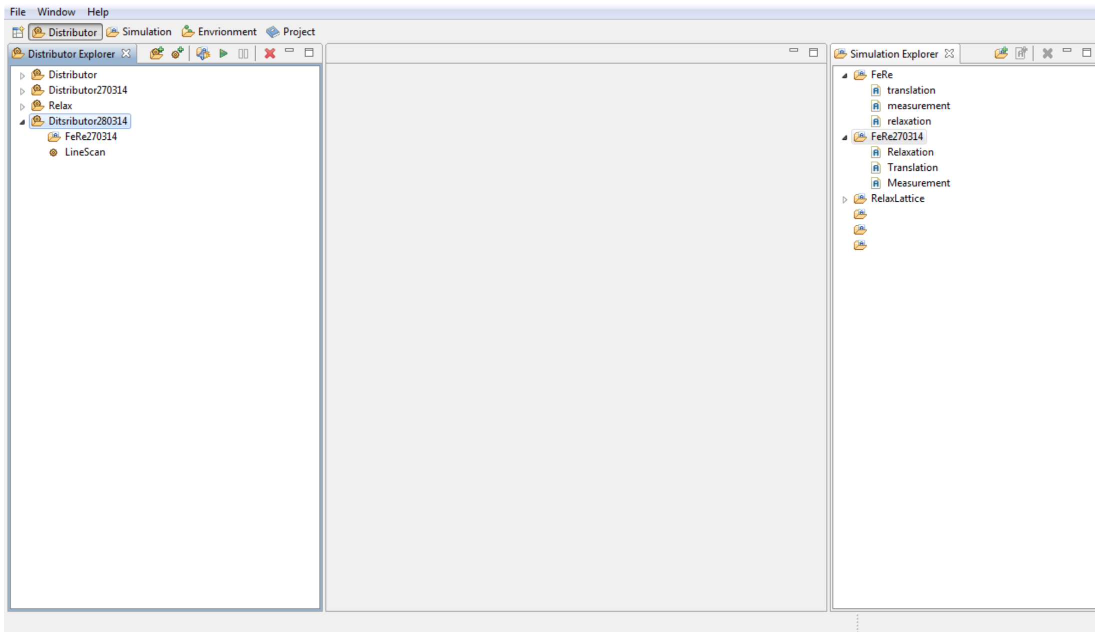

Abbildung A.4: Distributor-Perspektive für die Simulationsausführung und die Parallelisierung

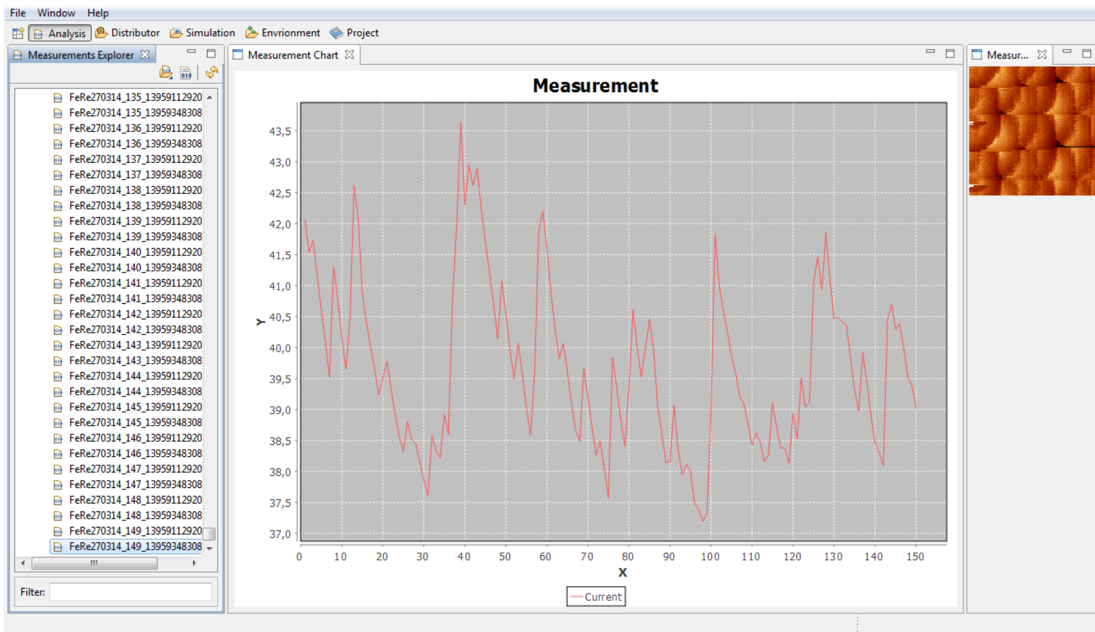

Abbildung A.5: Analysis-Perspektive zur Analyse der Ergebnisse

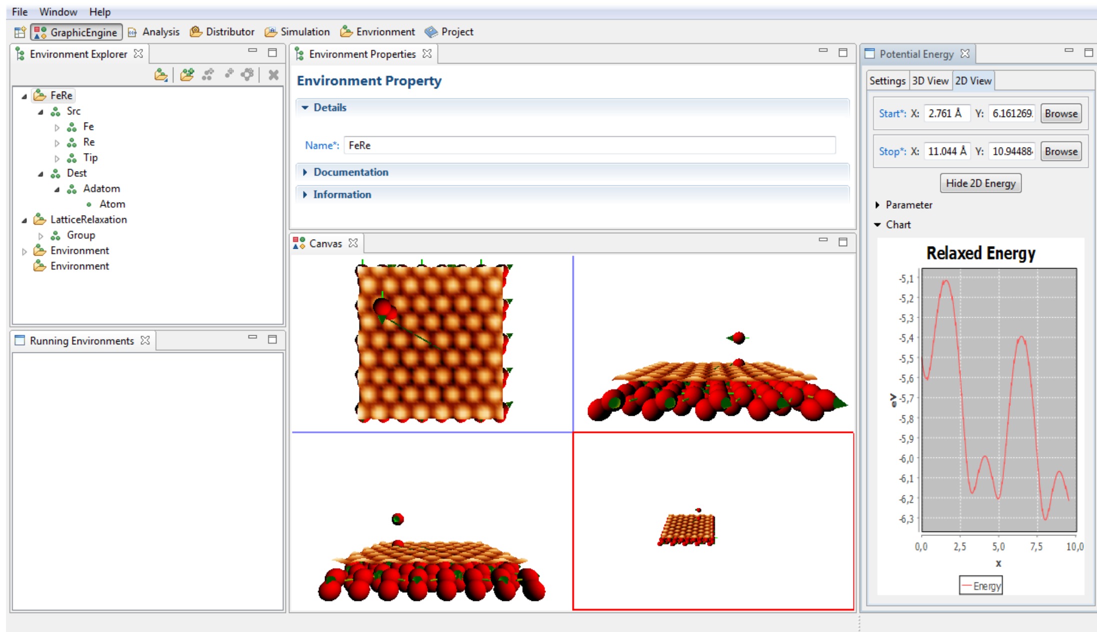

Abbildung A.6: Monitor-Perspektive zur Verifikation der Modellierung und Parametrisierung

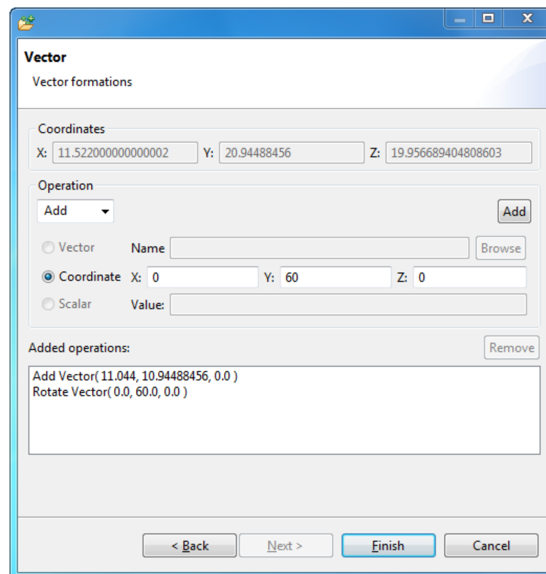

Abbildung A.7: Wizarddialog zum Erstellen eines Vector-Objekts

Literaturverzeichnis

[Binnig und Rohrer 1987] Binnig, Gerd ; Rohrer, H: Scanning tunneling microscopy-from birth to adolescence. In: *Reviews of Modern Physics* (1987), Nr. 1986. – URL http://www.phy.pku.edu.cn/~qhcao/resources/class/QM/RevModPhys.59.615.pdf

[Breu u. a. 2008] Breu, FX ; Guggenbichler, S ; Wollmann, JC: No Title. In: *Vasa* (2008). – URL http://medcontent.metapress.com/index/A65RM03P4874243N.pdf. ISBN 9783486590456

[Daum 2008] Daum, B: *Rich-client-Entwicklung mit Eclipse 3.3: Anwendungen entwickeln mit Eclipse RCP, SWT, Forms, GEF, BIRT, JPA u.a.m.* Dpunkt.Verlag GmbH, 2008. – ISBN 9783898645034

[Ebert 2011] Ebert, R: *Eclipse RCP - Entwicklung von Desktop-Anwendungen mit der Eclipse Rich Client Platform.* na, 2011

[Gamma u. a. 2011] Gamma, E ; Johnson, R ; Helm, R ; Vlissides, J: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Pearson Deutschland, 2011 (Programmer's choice). – 5–7 S. – ISBN 9783827330437

[Gross und Marx 2012] Gross, R ; Marx, A: *Festk{ö}rperphysik.* Oldenbourg Wissenschaftsverlag, 2012. – ISBN 9783486712940

[Hla 2005] Hla, Saw-wai: STM Single Atom / Molecule Manipulation and Its Application to Nanoscience and Technology. (2005), S. 1–12

[Hla u. a. 2003] Hla, SW ; Braun, KF ; Rieder, KH: How Single Atoms Move During a Quantum Corral Construction. In: *APS Meeting Abstracts* (2003), S. 2–5. – URL http://adsabs.harvard.edu/abs/2003APS..MARY15010B

[Khajetoorians u. a. 2012] Khajetoorians, Alexander A. ; Wiebe, Jens ; Chilian, Bruno ; Lounis, Samir ; Blugel, Stefan ; Wiesendanger, Roland: Atom-by-atom

engineering and magnetometry of tailored nanomagnets. In: *Nat Phys* 8 (2012), Juni, Nr. 6, S. 497–503. – URL http://dx.doi.org/10.1038/nphys2299http://www.nature.com/nphys/journal/v8/n6/abs/nphys2299.html#supplementary-information. – ISSN 1745-2473

[Kittel 2006] Kittel, C: *Einf{ü}hrung in die Festk{ö}rperphysik*. Oldenbourg, 2006. – URL http://books.google.de/books?id=b3L3f1BBavQC. – ISBN 9783486577235

[Kramer und Neculau 1998] Kramer, U ; Neculau, M: *Simulationstechnik*. Hanser, 1998. – URL http://books.google.de/books?id=z2KVMQEACAAJ. – ISBN 9783446192355

[McAffer u. a. 2010] McAffer, J ; Lemieux, J M. ; Aniszczyk, C: *Eclipse Rich Client Platform*. Pearson Education, 2010 (Eclipse Series). – URL http://books.google.de/books?id=fbxdpDTeELoC. – ISBN 9780321612342

[Mehl 1994] Mehl, H: *Methodenglish Verteilter Simulation*. Vieweg+Teubner Verlag, 1994 (Programm Angewandte Informatik). – 2–5 S. – ISBN 9783528054397

[Neelamkavil 1987] Neelamkavil, F: *Computer Simulation and Modelling*. Wiley, 1987. – ISBN 9780471911296

[Page und Häuslein 1991] Page, B ; Häuslein, A: *Diskrete Simulation: Eine Einführung mit Modula-2*. Springer Berlin Heidelberg, 1991 (Springer-Lehrbuch). – ISBN 9783540544210

[Sommerville 2007] Sommerville, I: *Software Engineering*. 9. Addison-Wesley, 2007 (International computer science series). – ISBN 9780321313799

[Toumey 2010] Toumey, Chris: 35 Atoms That Changed the Nanoworld. In: *Nature nanotechnology* 5 (2010), April, Nr. 4, S. 239–41. – URL http://www.ncbi.nlm.nih.gov/pubmed/20348911. – ISSN 1748-3395

[University Ulm 2014] University Ulm: Praktikum Physikalische Chemie für Fortgeschrittene Versuch 11 : Rastertunnelmikroskopie (STM) Kolloquiumsthemen Prinzip der Rastertunnelmikroskopie / University Ulm. Ulm, 2014. – Forschungsbericht. – URL http://www.uni-ulm.de/physchem-praktikum/media/fp/v_11.pdf

[Warschat und Wagner 1995] Warschat, Joachim ; Wagner, F: Einführung in die Simulationstechnik. (1995), S. 32. – URL http://www.iat.

uni-stuttgart.de/lehre/lehrveranstaltungen/skripte/simulation/AltesSimulationsSkript.pdf

[Wautelet u. a. 2003] Wautelet, Michel ; Beljonne, David ; Lazzaroni, Roberto ; Alexandre, Michael: Nanotechnologie. In: *Nanotechnologie* (2003), S. 84

[Wetzlmaier 2007] Wetzlmaier, T: *Systematische Testfallgenerierung f{ü}r den Black-Box-Test*. Bod Third Party Titles, 2007. – URL http://books.google.de/books?id=VD6cA5Uw7WAC. – ISBN 9783638680011

[Wolter 2009] Wolter, Boris: *Semi-autonome laterale Manipulation einzelner Atome innerhalb einer simulierten Rastertunnelmikroskopumgebung*, Universität Hamburg, Thesis, 2009. – URL http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Semi-autonome+laterale+Manipulation+einzelner+Atome+innerhalb+einer+simulierten+Rastertunnelmikroskopumgebung#0

[Wolter 2014] Wolter, Boris: *Magnetic Atom Manipulation and Spin-dependent Atomic Friction Investigated by Spin-polarized Scanning Tunneling Microscopy and Monte Carlo Simulations*, Universität Hamburg, Dissertation, 2014

[Wolter u. a. 2012] Wolter, Boris ; Yoshida, Yasuo ; Kubetzka, André ; Hla, Saw-Wai ; Bergmann, Kirsten von ; Wiesendanger, Roland: Spin Friction Observed on the Atomic Scale. In: *Physical Review Letters* 109 (2012), September, Nr. 11, S. 116102. – URL http://link.aps.org/doi/10.1103/PhysRevLett.109.116102. – ISSN 0031-9007

Glossar

Adatom In der Oberflächenphysik bezeichnet man mit Adatom (Adsorbed Atom) ein Fremdatom gebunden an eine Festkörperoberfläche. 12, 19, 21, 28–30, 60, 67

Black-Box-Test Teststrategie zum funktionsorientierten Testen von Komponenten bei denen die innere Funktionsweise nicht betrachtet wird. 64

DataBindig Synchronisation von Anzeigedaten einer grafischen Benutzerschnittstelle und dem Applikationskern. 39

deserialisieren Rekonstruktion von strukturierten Daten anhand einer sequenziellen Darstellungsform. 56

Listener Teil des Observer Patterns. Bietet eine Aktualisierungsschnittstelle für Objekte, die Änderungen ihres Zustands mitteilen. 35, 40

Persistiert Abspeichern von Daten auf einem nichtflüchtigen Speichermedium über einen längeren Zeitraum. 53

Regular Expressions Zur Beschreibung von Mengen von Zeichenketten mithilfe syntaktischer Regeln. Häufig eingesetzt bei Such-Problemen in Texten. 56

Seed Startwert für deterministische Zufallszahlengeneratoren zur Reproduktion der Zahlenfolge. 69

serialisiert Abbildung von strukturierten Daten auf eine sequenzielle Darstellungsform. 37

Singleton Das Singleton-Pattern erzeugt genau eine Objektinstanz einer Klasse und stellt einen globalen Zugriffspunkt bereit. 35, 50

Unit-Test Testmethode zum Überprüfen der Funktionsweise von Modulen. Üblicherweise wird zuerst das Modul in einen initialen Zustand versetzt, anschließend die zu testende Operation ausgeführt und zuletzt ein Soll-Wert mit dem Ist-Ergebnis verglichen. 64

Validator Als Teil des DataBinding-Konzepts überprüft der Validator die Daten. Erst nach einer erfolgreicher Überprüfung werden die Daten an das verbundene Attribut übergeben. 40

Validität Bezeichnet das argumentative Gewicht einer Aussage oder Untersuchung. 5

Viewpart Teil des RCP-UI. Bezeichnet ein Fensterelement, das innerhalb der Workbench angezeigt wird. 41, 47

Widget Element einer grafischen Benutzeroberfläche für die Interaktion mit dem Benutzer. 39, 55

Abkürzungsverzeichnis

RTM Rastertunnelmikroskop

BCC Body centered cubic

FCC Face centered cubic

HCP Hexagonal close packed

LM Laterale Manipulation

RCP Rich-Client-Platform

SWT Standard-Widget-Toolkit

MVC Model-View-Control

AWT Abstract-Window-Toolkit

Abbildungsverzeichnis

2.1 Vorgehensmodell in Simulationsstudien 5
2.2 Betriebsmodi eines RTMs 9
2.3 Häufig vorkommende Kristallgitter 9
2.4 Laterale Manipulation von Ag Atomen 11
2.5 Komponenten der RCP 13

3.1 Aufbau einer Komponente 21
3.2 Aufbau der Environment-Komponente 21
3.3 Aufbau der Simulationskomponete 24
3.4 Ablauf der Prozessmodellierung 26
3.5 Atommengen eines Systems 27
3.6 Aufbau der Distributor-Komponente 28
3.7 Aufbau der Monitor-Komponente 30
3.8 Aufbau der Analysis-Komponente 31

4.1 Übersicht der Systemkomponenten 33
4.2 Hinzufügen von Datenobjekten 36
4.3 Arbeitsweise des WizardStateHandlers. In Abhängigkeit des Eingabeelements werden die Fehlerbedingungen durchlaufen und überprüft 44
4.4 Interaktion zwischen View und Model Plug-Ins 50
4.5 Event-basierte Kommunikation der Aktionen 57

5.1 Aufbau einer TestUnit für den Black-Box Test des ActionSchedulers 64
5.2 Schnitt eines BCC Gitters durch die Ebene (110) 66
5.3 Bewegungspfad der Spitze über die Mn Lage 67
5.4 Antiparallele Ausrichtung der benachbarten Mn Reihen 68
5.5 Energie des Adatoms (rot) verglichen mit Ergebnissen von Wolter u. a. (2012) (grün) und der Abweichung in Prozent (blau) 69

A.1 Platform-Perspektive beinhaltet eine Übersicht aller erzeugten Datenobjekte und die Dokumentation der Software 73
A.2 Environment-Perspektive zur Modellierung des Systems 74
A.3 Simulation-Perspektive zur Modellierung des Prozesses 74
A.4 Distributor-Perspektive für die Simulationsausführung und die Parallelisierung 75
A.5 Analysis-Perspektive zur Analyse der Ergebnisse 75
A.6 Monitor-Perspektive zur Verifikation der Modellierung und Parametrisierung 76
A.7 Wizarddialog zum Erstellen eines Vector-Objekts 76

Quellcodeverzeichnis

4.1 Interface IDataHandler für die Datenverarbeitung im Model 34
4.2 Hinzufügen eines DataHandlerListener 37
4.3 Ereignissignalisierung in der Model Komponente 37
4.4 Hinzufügen eines DataListener 37
4.5 Ereignissignalisierung in dem Datenobjekt 38
4.6 Vorausgesetzte Operationen und Attribute für das DataBinding 39
4.7 Erzeugen eines Bindings zwischen View und Model 40
4.8 Aktualisierung eines Bindings durch Benutzerinteraktion 40
4.9 Fehlerbedingungen des WizardStateHandlers 42
4.10 Instantiierung und Initialisierung des WizardStateHandlers 43
4.11 Fehlerprüfung des WizardStateHandlers durch Benutzerinteraktionen 44
4.12 SelectionRestriction für die Auswahlbeschränkung von Dialogfenstern 45
4.13 Kommando Deklaration in der Plugin.xml 46
4.14 Image Deklaration für Commands 46
4.15 Deklaration von Menüstrukturen in der Plugin.xml 46
4.16 Zuweisung des Ausführungsverhaltens für einen Command 47
4.17 SelectionRestriction zur Aktivierung von Commands 48
4.18 Regeldeklaration für die Sichtbarkeit eines Kommandos 49
4.19 Algorithmus zur Gittererzeugung nach Bravais 55
4.20 MaskParameterSet für die Zuweisung von Parametern zur Laufzeit 58
4.21 Auswahlbeschränkung von ParameterSets über die MaskParameterSet 58

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

Hamburg, 9. Juli 2014 Florian Meyer