Spezifikation von regelgetriebenen Simulationen biochemischer Prozesse

Specification of rule-based simulations of biochemical processes
Bachelorarbeit im Studienbereich Information Systems Technology von Tobias Niehues
Tag der Einreichung: 3. April 2020

1. Gutachten: Prof. Dr. rer. nat. Andy Schürr
2. Gutachten: M. Sc. Sebastian Ehmes
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Studienbereich Information Systems Technology
Fachgebiet Echtzeitsysteme

Spezifikation von regelgetriebenen Simulationen biochemischer Prozesse
Specification of rule-based simulations of biochemical processes

Bachelorarbeit im Studienbereich Information Systems Technology von Tobias Niehues

1. Gutachten: Prof. Dr. rer. nat. Andy Schürr
2. Gutachten: M. Sc. Sebastian Ehmes

Tag der Einreichung: 3. April 2020

Darmstadt

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Tobias Niehues, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, den 3. April 2020

T. Niehues

Zusammenfassung

Die moderne Biochemie eröffnet uns die Perspektive, Krankheiten wie Krebs, Diabetes, Alzheimer oder andere Erkrankungen, in denen die zellregulatorischen Funktionen im Stoffwechsel eines Organismus versagen, besser zu verstehen und womöglich sogar zu bekämpfen. Dass dies überhaupt möglich ist, verdanken wir einer breiten Basis an fachspezifischem Wissen, die unter anderem durch rechnerbasierte Simulationen der entsprechenden Zell- und Enzyminteraktionen entstanden ist.
Diese Arbeit betrachtet vornehmlich die Simulation solcher Interaktionen via regelbasierter Ansätze. Hierbei wird das Verhalten komplexer biochemischer Prozesse eines Systems in einzelne, immer wiederkehrende Muster zerlegt, um es mittels Mustererkennungstools und entsprechender Transformationen der erkannten Muster vollständig zu modellieren und simulieren.
Existierende, etablierte Spezifikationen wie *Kappa* oder die *BioNetGenLanguage* ermöglichen es, die Modelle solcher Simulationen via domänenspezifischer Sprachen zu definieren. Diese bieten bereits umfangreiche Modellierungsoptionen, haben jedoch noch einige Probleme und ungenutztes Potenzial in Bezug auf das intuitive Sprachverständnis sowie ihre allgemeine Ergonomie respektive ihre Bedienbarkeit.
Im Hinblick auf diese Parameter wird eine Spezifikation solcher regelbasierten Simulationen und ein zugehöriges Framework zur Anbindung an das bestehende Simulationstool *SimSG* entworfen und implementiert.
Anschließend wird diese neue Sprache dahingehend evaluiert, ob die zur Entwicklung richtungsweisenden Aspekte optimiert werden konnten, und die verschiedenen zur Simulation genutzten Mustererkennungstools aufgrund verschiedener Modellstrukturen miteinander verglichen.

Abstract

Modern biochemistry opens new perspectives in understanding and finding remedies for diseases like cancer, diabetes or Alzheimer's, where regulatory mechanisms of cells in an organism's metabolism fail. This is made possible due to broad and highly specialized knowledge in biochemical contexts, obtained by computer-based simulations of diverse cell and enzyme interactions.
This work focusses on the simulation of such interactions via the rule-based method. Herein, the behavior of complex biochemical process in a system is split into several reoccuring patterns, to be completely modeled and simulated by the use of pattern matching tools and the according model transformations.
Already existent and well-established specifications such as *Kappa* or the *BioNetGenLanguage* provide extensive possibilities to model such systems and simulations employing domain-specific languages. Still, these have issues in terms of their intuitive comprehensibility and general usability nonetheless. In regard to those parameters a specification for such rule-based simulations and a corresponding framework for integrating it into the already existent simulation tool *SimSG* is developed and implemented.
Finally this new language is evaluated with respect to the intended optimization of the given aspects and the pattern matching tools used for simulation are compared based on different models of various types.

Inhaltsverzeichnis

1. Einleitung

Die moderne Biochemie ermöglicht es, Stoffwechselprozesse in Organismen auf molekularer Ebene zu erforschen und verstehen. Fortschritt in diesem Forschungsgebiet bildet das Fundament zur Bekämpfung von beispielsweise Krebs oder Diabetes bis hin zu weiteren Erkrankungen, in denen die Funktionen und Interaktion von Zellorganismen gestört sind. Hierbei ist präzises Wissen über die Signaltransduktion zwischen und innerhalb verschiedener Zellen obligatorisch.
Um neues Wissen dieser Art zu erwerben oder schon bestehende Theorien zu verifizieren, werden seit jeher Simulationen biochemischer Prozesse durchgeführt. Zur Realisierung dieser Simulationen wurden über die Jahre mehrere Methoden entwickelt, besonders im Vordergrund stehen jedoch die Modellierung mittels Systemen gewöhnlicher Differentialgleichungen (*ordinary differential equation*, ODE) und regelbasierte Ansätze.
Beide Varianten haben sich durchaus bewährt, jedoch erfordert das Aufstellen solcher ODE-Systeme hochspezifisches Wissen über die komplexen Vorgänge innerhalb der betrachteten biochemischen Komponente. Regelbasierte Ansätze hingegen schlagen einen anderen Weg ein. Hierbei wird das Abstraktionslevel erhöht, was die Modellierung dieser Vorgänge sowohl weniger aufwändig und fehleranfällig, als auch leichter zugänglich macht. Jedoch sind sie auch weniger akkurat.
Diese regelbasierten Modelle müssen allerdings auch effizient erstellt werden können, sodass der User präzise formulieren kann, was er abbilden möchte, damit die Simulatoren dies entsprechend umsetzen. Hierzu bieten sich domänenspezifische Sprachen (*domain specific language*, DSL) an, die konkret für ihren Anwendungskontext entwickelt werden und in diesem hochspezialisiert sind. Etablierte Vertreter der regelbasierten Modellierung in der Biochemie-Domäne sind Sprachen wie die *BioNetGenLanguage* (BNGL) und *Kappa*[1] respektive die zugehörigen regelbasierten Simulationstools *BioNetGen*[1] [2] und *KaSim*[2].
Diese Werkzeuge arbeiten zwar effizient in der Simulation biochemischer Prozesse, bieten jedoch kaum Raum zur Formulierung komplexer Anwendungsbedingungen für Regeln. Das Simulationstool *SimSG*[4] verwendet universale Graphtransformationswerkzeuge wie im *CASE*-Tool *eMoflon*[3] [3], welche zwar nicht für die entsprechende Problemdomäne optimiert sind, aber durch die Verwendung von universalen Graphmustererkennungswerkzeugen beispielsweise die Definition komplexer Vorbedingungen und Programmierung diverser Regelanwendungsstrategien erlauben.

[1] `www.csb.pitt.edu/Faculty/Faeder/` [2] `github.com/Kappa-Dev/KaSim` [3] `emoflon.org/`

1.1. Ziel der Thesis

Ziel dieser Thesis ist nun die Entwicklung eines Frameworks zur kompakten Spezifikation biochemischer Reaktionsprozesse, mit dem regelbasierte Simulationen mit Hilfe von *eMoflon* und dem darauf basierenden Simulationswerkzeug *SimSG*[4] durchgeführt werden können.
Hierzu wird zunächst eine entsprechende domänenspezifische Sprache entwickelt. Mit dieser DSL werden Modelle beschrieben, die biochemische Entitäten, deren Interaktion mittels Reaktionsregeln und gewisser Vorbedingungen, sowie weitere Modelleigenschaften spezifiziert.
Die DSL wird sich hierbei an bereits existierenden Ansätzen orientieren und deren beste Aspekte zusammenführen. Im Fokus stehen hierbei präzise Formulierungen und eine umfangreiche Ausdrucksstärke, ohne dass darunter die Kompaktheit oder Intuitivität der Sprache leidet. Die Syntax der Sprachgrammatik muss mit Sorgfalt gewählt werden, damit sie auch für Anwender aus dem Biochemiekontext, die noch keine Erfahrung mit domänenspezifischen oder Programmiersprachen haben, leicht verständlich und nutzbar ist.
Des Weiteren wird eine Transformation der via DSL entstehenden Spezifikationsmodelle implementiert, um das beschriebene Modell auch für *general-purpose Tools* - hier konkret *eMoflon* - interpretierbar zu machen. Dazu werden die Reaktionsregeln via eines allgemeinen Zwischenmodells übersetzt und aus den Modellspezifikationen der Sprache konkrete Modellinstanzen zur Simulation generiert. Diese Instanzen liegen in Form von graphbasierten Modellen vor, sodass die formulierten Regeln mittels der in eMoflon integrierten Mustererkennungswerkzeuge (*pattern matching tools*) umgesetzt werden können.
Abschließend erfolgt eine Evaluation anhand verschiedener biochemischer Reaktionsprozesse im Kontext des entwickelten Frameworks und ein Vergleich der in eMoflon enthaltenen Pattern-Matching-Werkzeuge im Hinblick auf verschiedene Parameter wie Modellumfang, Laufzeit und Speicherverbrauch.

1.2. Thesis-Struktur

Die Thesis beginnt in Kapitel 2 mit der Einführung aller für das Verständnis der Arbeit nötigen theoretischen Grundlagen. Hierzu wird in Abschnitt 2.1 die Funktionsweise biochemischer Prozesse grob skizziert und in Teil 2.1.1 anhand eines praktischen Beispiels veranschaulicht.
Abschnitt 2.2 gibt einen Überblick darüber, wie Prozesse dieser Art rechnerbasiert simuliert werden können und stellt die beiden prominentesten Methoden über Differentialgleichungssysteme und via einer endlichen Menge an Reaktionsregeln vor (2.2.1 und 2.2.2). Zudem werden mögliche Spezifikationen des letzteren Ansatzes mitsamt etablierter Vertreter vorgestellt (2.2.3).
Das Kapitel schließt mit der Beleuchtung verschiedener Aspekte des *Model-Driven Engineering* (MDE) in Abschnitt 2.3. Dazu zählen Klassifikationen und Eigenschaften diverser Modelltypen, sowie *domänenspezifische Sprachen* (2.3.1), Modelltransformationen und deren Kategorisierung (2.3.2) als Werkzeuge des MDE. Da *Pattern Matching* für die hier genutzten Simulationsmethoden unerlässlich ist und in dieser Arbeit primär auf graphenbasierten Modellen angewandt wird, werden diese hier ebenfalls kurz eingeführt (2.3.3).

Kapitel 3 stellt den Kern dieser Arbeit dar und präsentiert die Implementierung des Frameworks. Dazu wird zunächst in Abschnitt 3.1 eine kurze Übersicht über die relevanten Module gegeben, die für das Framework entwickelt wurden und wie diese in das bereits vorhandene Simulationstool *SimSG*

integriert wurden. Im Anschluss werden kurz die für die Modellierung und Implementierung genutzten Tools in Abschnitt 3.2 vorgestellt.
Der Entwicklungsprozess der neuen Sprache *Re.action* wird in Abschnitt 3.3 mit den wegweisenden Leitlinien des Designs (3.3.1) und den daraus resultierten Features und Möglichkeiten (3.3.2) dargelegt.
Zuletzt beschreibt 3.4 die Integration und Anbindung des Frameworks an *SimSG*, indem das aus der Sprache resultierende Modell der Spezifikation (3.4.1) und dessen Überführung via verschiedener Transformationen über ein weiteres Zwischenmodell (3.4.2) hin zu einem ausführbaren Simulationsmodell erläutert wird (3.4.3).

Um die Ergebnisse und den Nutzen der Implementierung zu bewerten, erfolgt in Kapitel 4 sowohl eine Evaluation der neuen Spezifikation *Re.action*, als auch der in *SimSG* genutzten Pattern Matcher.
Im Zuge dessen wird in Abschnitt 4.1 zunächst die allgemeine Nutzbarkeit und Benutzerfreundlichkeit der *Re.action*-Sprache per se bewertet (4.1.1) und daraufhin auch im Vergleich mit anderen etablierten Spezifikationen wie *Kappa* und *BioNetGen* betrachtet (4.1.2). Im Anschluss erfolgt eine kurze Zusammenfassung der Ergebnisse (4.1.3).
Abschnitt 4.2 bildet die zweite Hälfte der Evaluation und beschäftigt sich mit der Performance der in *SimSG* respektive *eMoflon* integrierten Pattern-Matching-Engines in Bezug auf ihre Laufzeit und ihren Speicherverbrauch. Der Aufbau der Evaluation mit den technischen Daten des Testgeräts und den zur Simulation verwendeten Modellen wird in 4.2.1 beschrieben und abschließend werden die Ergebnisse der Simulationen in 4.2.2 diskutiert.

Einen Auszug des aktuellen *state of the art* der Spezifikation und Simulation biochemischer Systeme und Prozesse bietet Kapitel 5. Hierzu wird eine Auswahl der prominentesten Vertreter getroffen, welche in Abschnitt 5.1 mit dem *Kappa*-Framework begonnen wird. in den Abschnitte 5.2 und 5.3 folgen die *Systems Biology Markup Language* (SBML)[5] und das *BioNetGen*-Framework (BNG)[2]. Zuletzt werden mit *RuleBender*[6] und *CellDesigner*[7] in den Abschnitten 5.4 und 5.5 zwei Tools zur Erstellung und Analyse biochemischer Modelle vorgestellt, die diese Prozesse mittels Visualisierungen unterstützen.

Am Schluss dieser Arbeit wird in Kapitel 6 diskutiert, ob und wie die gestellten Ziele erreicht werden konnten. Hierzu werden auch die Ergebnisse der Evaluation miteinbezogen und in Abschnitt 6.1 die zukünftigen Perspektiven des entwickelten *Re.action*-Frameworks hervorgehoben.

2. Grundlagen

Die Entwicklung einer Spezifikation für biochemische Prozesse setzt ein grundlegendes Verständnis für Reaktionen und Prozesse dieser Art voraus.
Die dafür nötigen Grundlagen, wie Eigenschaften und Ablauf solcher biochemischer Reaktionen werden deshalb in Kapitel 2.1 erklärt und an einem praktischen Beispiel veranschaulicht.
Da das hier implementierte Framework für regelbasierte Simulationen entwickelt wird, geht Kapitel 2.2 auf die etablierten Simulationsmethoden der Biochemiedomäne ein und in Abschnitt 2.2.2 speziell auf diese regelbasierten Ansätze. Insbesondere geht es hierbei auch um die Spezifikation von Modellen für diese Art von Simulation. Diese wird anhand des bereits bekannten κ-Kalküls[8] zur Modellierung solcher Ansätze, auf dem auch die hier entwickelte Sprache aufbaut, näher ausgeführt.
Solche Simulationen arbeiten auf bestimmten, durch die Spezifikation definierten Modellen verschiedenster Formen. Hierzu stellt Kapitel 2.3 die Grundlagen der modellgetriebenen Softwareentwicklung vor, indem es domänenspezifische Sprachen und Modelltransformationen einführt. Speziell in diesem Kontext handelt es sich um graphenbasierte Modelle, weswegen Abschnitt 2.3.3 abschließend Graphmustererkennung näher beleuchtet.

2.1. Biochemische Prozesse

Chemische Prozesse und Reaktionen finden permanent in den Zellen von Lebewesen statt. Hierbei werden die chemischen Verbindungen von Molekülen untereinander manipuliert, wobei entweder Energie freigegeben oder aufgenommen wird. Formal können diese Prozesse über Reaktionsgleichungen beschrieben werden. Diese Gleichungen bestehen aus Reaktanten, die eine Menge von verschiedenen biochemischen Komponenten vor Ablauf der Reaktion, und Produkten, die eine Menge verschiedener biochemischer Komponenten nach Ablauf der Reaktion beschreiben und aus den Reaktanten heraus entstehen. Eine einfache chemische Reaktion könnte also beispielsweise wie die Knallgasreaktion in Gleichung 2.1 aussehen.

$$2H_2 + O_2 \longrightarrow 2H_2O \tag{2.1}$$

Wenn man die Wechselwirkung dieser Moleküle nun innerhalb eines biologischen Kontexts – also einem konkreten Organismus – betrachtet, befindet man sich in der Biochemie.
Ziel der Biochemie ist es somit, zu verstehen, wie Biomoleküle – also Proteine, Nucleinsäuren, Lipide etc. – in den Zellen von Organismen miteinander interagieren. Hierbei bezeichnet man die Konfigurationen verschiedener solcher Biomolekülkomplexe als *Spezies*. Durch eine Verkettung von solchen Reaktionen entstehen sogenannte Signalwege. Hierbei werden durch ständigen Zerfall, Manipulation und Neubildung dieser Biomoleküle Signale durch den Organismus getragen (*Signaltransduktion*).
Alle Verbindungen, die unter den verschiedenen Komponenten eines Moleküls eingegangen werden, bestehen zwischen sogenannten Sites. Sie sind quasi die Andockstellen für weitere Biomoleküle. Zudem besitzen sie eigene Zustände. Diese beschreiben den Status der Site und können zum Beispiel

durch verschiedene Vorgänge wie Phosphorylierung bzw. Ubiquitinierung manipuliert werden. Hierbei werden bestimmte funktionelle Gruppen – also Molekülgruppen, die die wesentlichen (Reaktions-)Eigenschaften einer Verbindung beeinflussen – wie zum Beispiel eine sogenannte *Phosphorylgruppe*, die aus einem Phosphor- und drei Sauerstoffatomen besteht, bzw. das Protein *Ubiquitin* an die Site gebunden. Auf diese Art und Weise wird eine Kommunikation von Zellen innerhalb eines Organismus untereinander möglich. Ein umfassendes Verständnis für den Ablauf dieser Signalwege hilft, die genauen Funktionen verschiedener Enzyme zu erkennen und somit auch Einblicke in die Entstehung und Verläufe von Krankheiten zu erhalten, die durch molekulare Störungen innerhalb solcher Signalwege verursacht werden.

2.1.1. Rolle von Zellzyklus kontrollierenden Enzymen in der Entstehung von Alzheimer

Eine weit verbreitete neurodegenerative Erkrankung, bei der die zellregulatorischen Eigenschaften des menschlichen Körpers versagen, ist *Alzheimer*. Die Patienten leiden unter Gedächtnis- und Orientierungsverlust, da die Verbindungen zwischen den Nervenzellen im Gehirn und sogar ganze Zellen selbst absterben.
Proctor und Gray untersuchten 2010 Zusammenhänge in den Signalwegen der Enzyme *Glykogensynthase-Kinase 3 Beta* und *p53* als treibende Faktoren für die Entwicklung und das Voranschreiten der Krankheit [9].
Glykogensynthase-Kinase 3 Beta (GSK3β) spielt eine Rolle in vielen physiologischen Stoffwechselprozessen und auch der *Apoptose* von Zellen – ein Selbstverteidigungsmechanismus des Körpers, bei dem kranke oder mutierte Zellen sich selbst abtöten, um weitere Schäden im Organismus zu vermeiden. *GSK3β* wurde bereits in verschiedenen Formen von Alzheimer als auffällige Komponente beobachtet und *Hooper et al.* formulierten die Hypothese, dass Überaktivität dieses Enzyms die Produktion von *Beta-Amyloiden (Aβ)* ankurbelt, das eigentlich eine Rolle bei der Informationsverarbeitung im Gehirn einnimmt, aber bei hohem Aufkommen zu sogenannten *Plaques* verklumpt und die Nervenbahnen im Gehirn im weitesten Sinne „verstopft“ [10]. Die am untersuchten Prozess beteiligten Enzyme sowie deren Interaktion und Bindungen untereinander zeigt Abbildung 2.1.
Das Protein *p53* ist eigentlich ein Tumorsuppressor, indem es Zellen unterdrückt, bei denen eine DNA-Schädigung vorliegt. Es ist genau wie *GSK3β* im Apoptose-Mechanismus enthalten, ist allerdings die treibende Kraft, die geschädigte Zellen letztendlich wirklich in den Zelltod schickt. Aus dem Grund ist es wichtig, dass die im Körper aktive Menge *p53* stets niedrig gehalten wird und nur bei auftretenden DNA-Schäden kurzzeitig und geringfügig erhöht wird, da sonst die Gefahr besteht, dass versehentlich zu viele Zellen abgetötet werden. Dies passiert bei gesundem Stoffwechsel und solange keine geschädigten Zellen vorliegen durch Bindung von *p53* an das Enzym *Mdm2*. Durch diese Bindung an *Mdm2* wird das *p53* zur Degradation – also zur Zersetzung – markiert. Das auf diese Weise markierte *p53* wird dann durch sogenannte *Proteasome* abgebaut. Dies sind Enzyme, die speziell dem Abbau anderer Proteine dienen.
Hooper et al. beobachteten, dass es bei Alzheimer zu einer erhöhten Konzentration von *p53* im Körper kommt und dies zu einer starken Phosphorylierung (*Hyperphosphorylierung*) von *Tau*-Enzymen kommt [11], die eigentlich für die Stabilität und Nährstoffversorgung der Zellen verantwortlich sind. Nun sammelt es sich in den Nervenzellen in Form von sogenannten *Tau-Fibrillen*, wodurch diese Zellen ihre Form und Funktion verlieren, bis sie schließlich gänzlich absterben.
Proctor und Gray untersuchten nun die Hypothese, dass diese Hyperphosphorylierung bei Alzheimer durch Interaktion von *p53* und *GSK3β* entstehe, sowie *GSK3β* und *p53* in eine positive Rückkopplungsschleife gerieten, die zu der von *Hooper et al.* beschriebenen erhöhten Aktivität von *p53* führe.

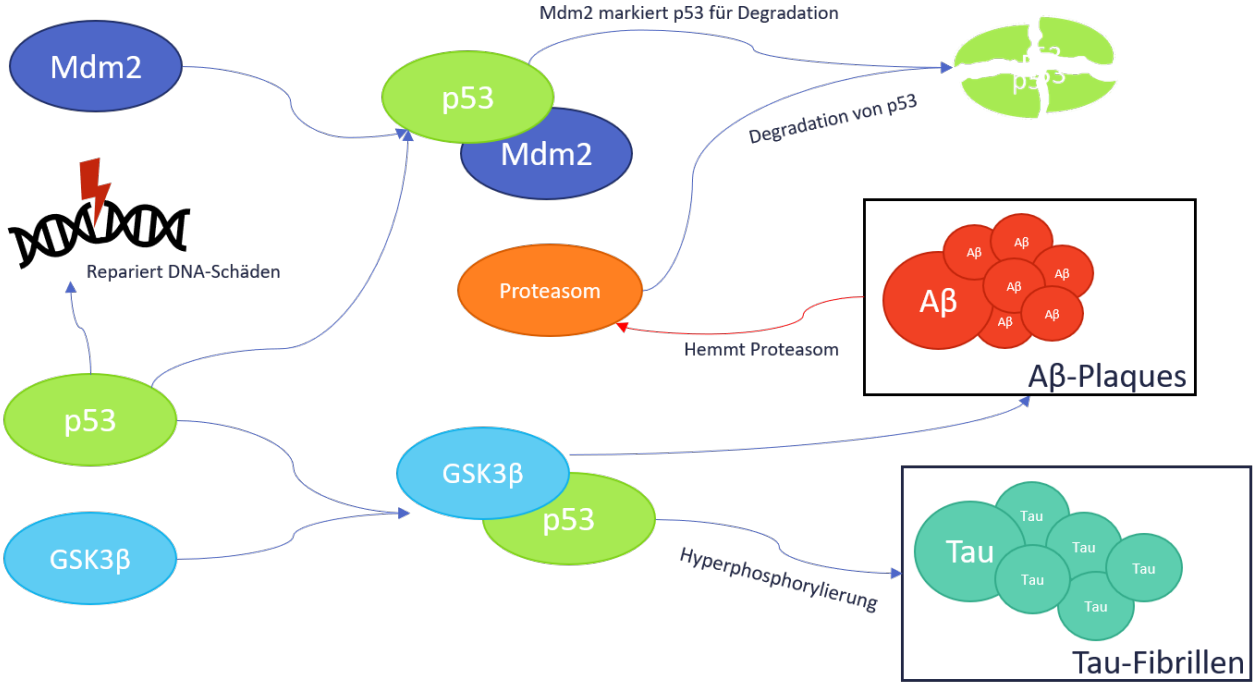


Abbildung 2.1.: Interaktion der involvierten Enzyme untereinander. Wird das *Proteasom* gehemmt, erhöht sich die Konzentration von *p53*, da es nicht mehr durch das *Proteasom* abgebaut wird. Durch Interaktion von *p53* und *GSK3*β entstehen nun noch mehr *A*β-Plaques, wodurch die Hemmung des Proteasoms weiter verstärkt wird.

Die Funktion der normalerweise für den Abbau verschiedener Proteine zuständigen *Proteasome* kann zum Beispiel im Alter nachlassen, wodurch die proteasomale Degradation nicht mehr häufig genug durchgeführt wird und nicht mehr genug *p53* abgebaut werden kann. Die sonst gering gehaltene Menge an *p53* im Organismus erhöht sich und die Unterdrückung des *p53*-Levels wird eventuell instabil. Da nun mehr Einheiten von *p53* im Organismus bestehen, führt dies wiederum zu mehr Verbindungen zwischen *p53* und *GSK3*β, die miteinander eingegangen werden können. Durch die so angeregte Aktivität von *GSK3*β entstehen nun noch mehr der Alzheimer-typischen *Tau-Fibrillen* und *A*β-Plaques. Die *Plaques* aus *Beta-Amyloiden* hemmen nun erneut die sowieso bereits unterdrückten Proteasome, wodurch nochmals weniger *p53* abgebaut wird und der *p53*-Level im Körper weiter steigt. Nun beginnt der Zyklus wieder von vorne, da wieder mehr *p53* an die so aktivierten *GSK3*β-Enzyme binden, wodurch wieder mehr Plaques und Tau-Fibrillen entstehen, und so weiter. Zur Verifizierung dieser Überlegungen formulierten *Proctor und Gray* ein (auf etwas vereinfachten Annahmen basierendes) Modell, bei dem Infrarot-Strahlung als zusätzliche chemische Spezies modelliert wird, um experimentelle DNA-Schäden zu simulieren.

2.2. Simulation biochemischer Prozesse

Um Theorien zum Verhalten biochemischer Abläufe zu verifizieren, den Einfluss verschiedener Medikamente auf den Stoffwechsel eines Organismus zu prüfen oder um ganz neue Erkenntnisse über die Signalwege verschiedener Proteine zu gewinnen, sind präzise Simulationen essenziell. Diese sollen die Frage beantworten, wie sich die Konzentrationen von homogen verteilten, verschiedenen chemischen Spezies innerhalb eines festen Volumens über den Verlauf der Zeit ändern. Dazu erhält jede Spezies eine initiale Startzahl an Molekülen und verschiedene chemische Reaktionen, die das Zusammenspiel der Moleküle untereinander beschreiben.
Zur Art und Weise, wie dieses Verhalten modelliert und simuliert wird, gibt es nun verschiedene Ansätze. In diesem Kapitel soll Abschnitt 2.2.1 den Ansatz über gewöhnliche Differentialgleichungssysteme näher bringen und Abschnitt 2.2.2 erläutern, wie man eine solche Simulation mittels verschiedener Regeln realisieren kann.

2.2.1. Ansatz über Differentialgleichungssysteme

Der traditionelle Weg zur Simulation von gekoppelten chemischen Reaktionen besteht aus einem Ansatz über gewöhnliche Differentialgleichungssysteme. Hierbei wird jeder im betrachteten Volumen enthaltenen Spezies eine kontinuierliche, skalare Funktion zugeschrieben, die die Menge an Molekülen dieser Spezies zu jedem Zeitpunkt definiert. Nimmt man hierzu noch an, dass es sich bei den Reaktionen um Prozesse handelt, die mit einer kontinuierlichen Rate ablaufen, so lässt sich ein System aus gekoppelten, gewöhnlichen Differentialgleichungen erster Ordnung aufstellen.
Allerdings weist dieser Ansatz diverse Probleme auf, wie *Gillespie* bereits 1977 erkannt hat [12]. Erstens ließe sich die Anzahl von Molekülen nicht durch einen kontinuierlichen Wert beschreiben, da diese nur in diskreten Mengen auftreten können. Zudem handele es sich bei der zeitlichen Entwicklung biologischer Systeme nicht um deterministische Vorgänge, da Position und Impuls der einzelnen Moleküle niemals alle gleichzeitig und mit ausreichend präziser Genauigkeit bekannt sein könnten (vgl. *Heisenberg'sche Unschärferelation*).

Kombinatorische Komplexität Des Weiteren erklärten bereits *Hlavacek et al.*, dass Modelle diesen Ansatzes für komplexere Systeme nicht mehr handhabbar sind [13]. Wie bereits in Abschnitt 2.1 erwähnt, besitzen Proteine mehrere Sites, die sich in verschiedenen Zuständen befinden können. Da jede molekulare Spezies eine eigene Gleichung im Gleichungssystem erhält, generiert ein Protein mit 6 Sites bereits 64 Gleichungen. Betrachtet man nun sogar einen einfachen Homodimer, der aus zweien solcher Proteine besteht, erhält man mehr als 4000 Gleichungen nur für diesen einfachen Komplex.
Diese kombinatorische Explosion aus modifizierbaren Komponenten der einzelnen Proteine führt dazu, dass die schiere Menge an Gleichungen nicht mehr gehandhabt werden kann. Für sehr komplexe systembiologische Phänomene ist dieser traditionelle Ansatz also nicht mehr geeignet.

2.2.2. Regelbasierte Ansätze

Um dieser kombinatorischen Explosion entgegenzuwirken, etablierte sich ein neuer Ansatz: *Regelbasierte Modelle* [14].
Durch die Reduktion der implizit in einem Differentialgleichungssysteme enthaltenen ablaufenden (Teil-)Reaktionen in einem biochemischen System auf eine feste Menge aus Regeln immer wiederkehrender Muster, wird der Umfang etwas gebändigt. Dies geschieht jedoch auf Kosten der Akkuratesse. Auf Differentialgleichungssystemen basierende Ansätze führen in der Regel zu genaueren Ergebnissen und sind vor allem präzise reproduzierbar, da die Lösung einer Differentialgleichung ein deterministischer Prozess ist, während regelbasierte Ansätze nicht-deterministisch sind, da sie mit stochastischen Methoden arbeiten.
Diese regelbasierten Ansätze basieren auf der Annahme, dass sich Proteine und deren Interaktion untereinander als nebenläufige Prozesse beschreiben lassen und somit das gesamte Netzwerk aus Signalwegen als massiv verteiltes System betrachtet werden kann. Dies baut grundlegend auf der Idee von *Milners π-Kalkül* auf [15], welches normalerweise zur Beschreibung von Systemen mit nebenläufigen Prozessen, die via bestimmter Kanäle miteinander kommunizieren, genutzt wird.
So könnte man mit dem *π-Kalkül* unter anderem ein Netzwerk aus verteilten Systemen oder auch einfache Client-Server-Interaktionen modellieren. Stark vereinfacht würden hierbei der Server-Task und der Aufruf dieses Tasks durch einen Client als einzelne Prozesse modelliert werden, die eine endliche Menge an Kommunikationskanälen zur Übermittlung von Daten erhalten.
Konkret im Kontext der Biochemie sind diese Prozesse die verschiedenen an einem Signalweg beteiligten Moleküle, die wiederum weitere Prozesse besitzen, welche die Sites eines Moleküls repräsentieren, an denen sich Moleküle potenziell miteinander verbinden können. Diese werden nun als parallel verlaufend modelliert, da Bindungen dieser Strukturen untereinander und die allgemeine Aktivität der einzelnen Moleküle alle gleichzeitig ablaufen können. Die Etablierung und Auflösung von Verbindungen erfolgt nun über die Kommunikationskanäle der als Prozess modellierten Moleküle[16].
Regelbasierte Ansätze entfernen sich von dem Gedanken, alle molekularen Spezies einzeln modellieren zu wollen. Stattdessen wird die in der Biologie stärker etablierte Variante genutzt, nicht jedes Protein mit Sites in einer anderen Kombination aus Zuständen als eigene Spezies zu betrachten. Stattdessen spricht man von beliebigen an Reaktionen beteiligten Komponenten als *Agenten*, die auch immer dieselben Agenten einer bestimmten Spezies bleiben, deren Zustände jedoch variabel sind und sich beliebig ändern können. So kann man beliebige Kombinationen von Zuständen modellieren, ohne dass dazu jeweils eine komplett neue chemische Spezies eingeführt werden muss. Dies reduziert die Menge der im System vorhandenen Spezies ungemein. Das Ergebnis einer solchen Betrachtung eines molekularen Systems ist ein Graph, der alle Entitäten als Knoten enthält und die Verbindung der Agenten untereinander als Kanten zwischen verschiedenen Sites darstellt.

Zudem werden die Reaktionen der Agenten miteinander nicht mehr über einzelne Differentialgleichungen für die jeweiligen Spezies beschrieben, sondern durch einfach aufgebaute Regeln. Diese sind derart strukturiert, dass sie Vorbedingungen und Nachbedingungen für bestimmte Molekül-Muster formulieren. Die Vorbedingungen beschreiben hierbei, auf welche auftretende Verbindung aus Agenten und Sites die Reaktion angewandt werden soll, und die Nachbedingungen spezifizieren das Ergebnis der Reaktion als Muster, das den Bindungszustand der betroffenen Agenten hinterher beschreibt. Dabei beschreiben Vorbedingungen nicht zwingend ganze Moleküle, sondern eventuell auch nur die relevanten Teile von diesen. Tritt ein bestimmter Komplex als Teil eines Moleküls auf, der bei Finden einer bestimmten funktionellen Molekülgruppe phosphoryliert werden soll, so formuliert die Regel diese Phosphorylierung für alle Moleküle, die in irgendeiner Art und Weise den spezifizierten Komplex enthalten. Dies ist sehr praktisch, da das dynamische Verhalten verschiedener Komponenten konkreter beschrieben wird und ein einzelner Reaktionsschritt, der in vielen Reaktionen enthalten ist, in einer einzelnen Regel formuliert wird. Durch die Gesamtheit aller Regeln werden so implizit alle auftretenden Reaktionen modelliert.
Ebenso ist es auch intuitiver, Regeln statt Reaktionen zu formulieren, da diese deklarativ die Vorgänge verschiedener Prozesse beschreiben. Die Semantik einer mathematischen Differentialgleichung zu interpretieren fällt weitaus schwerer.
Des Weiteren ist die Schreibweise solcher Regeln nicht unbekannt. Ein Beispiel für Regeln dieser Art aus dem Modell von *Proctor und Gray* [9] ist die in Abbildung 2.2 gezeigte. Diese Regel beschreibt die

Reaktionsname	Reaktanten		Produkte	Reaktionsrate
GSK3β_p53Binding:	GSK3β, p53	$\rightarrow$	GSK3β+p53	$2.0 \cdot 10^{-6}\mathrm{molecule}^{-1}\mathrm{s}^{-1}$

Abbildung 2.2.: Reaktion zur Entstehung einer Verbindung zwischen *GSK3β* und *p53*

Entstehung einer Bindung zwischen den beiden Molekülen *GSK3β* und *p53*. Eben jene Moleküle treten in dieser Regel nun als Agenten auf. Für eine vollständige regelbasierte Simulation wird hier eigentlich auch noch eine Definition der Sites benötigt, über die die Agenten verbunden sind. Falls diese Sites im ursprünglichen Modell nicht vorhanden sind, da sie entweder physikalisch real nicht existieren oder es für das rein biochemische Modell nicht relevant ist, müssen diese zur programmatischen Simulation extra konstruiert werden. Zudem gehört zu jeder Regel eine Reaktionsrate, die beschreibt, wie oft die Reaktion pro beteiligtem Molekül abläuft.
Es existieren bereits viele Simulationstools und Spezifikationen, die solche regelbasierten Ansätze implementieren. Zu den prominentesten Vertretern dieser Anwendungsdomäne zählen unter anderem *BioNetGen*[1] (mitsamt seiner Sprache *BioNetGenLanguage* (BNGL)) [2] und *KaSim*[2] als Simulator zur *Kappa-Language* [1]. Die hier entwickelte Spezifikation soll die bisherige Sprache des Simulationstools *SimSG*[4] ersetzen.
Die meisten der genannten Simulationen basieren auf *Gillespies Algorithmus* zur Simulation von (bio-)chemischen Prozessen in einem festen Volumen [12] [17]. Die Grundidee hinter Gillespies „*stochastic simulation algorithm*“ ist es, die chemischen Reaktionen nicht als kontinuierlich und deterministisch, sondern als diskrete und stochastische Prozesse zu verstehen. Hierbei werden Moleküle auch wie zuvor erläutert als *Agenten* modelliert, die das physikalische Verhalten, wie z.B. die Position des Moleküls, abbilden. Zudem wird die Simulation in einzelne, diskrete Schritte aufgeteilt, wobei in jedem Schritt die Wahrscheinlichkeiten für die verschiedenen durch Kollisionen von Molekülen ausgelösten Reaktionen berechnet werden. Ob bestimmte Prozesse tatsächlich ablaufen, wird nun ausschließlich aufgrund dieser Wahrscheinlichkeiten entschieden.

[1] `www.csb.pitt.edu/Faculty/Faeder/` [2] `github.com/Kappa-Dev/KaSim`

2.2.3. Spezifikation regelbasierter Ansätze

Um solche Modelle und deren Prozesse zu simulieren, müssen diese jedoch auch detailgetreu und eindeutig definiert werden können. Hierzu bedarf es spezieller Spezifikationen, die am besten auf den biochemischen Anwendungskontext zugeschnitten sind. Die Sprache *Kappa*[1] hat sich hierbei mit ihrem zugrunde liegenden κ-*Kalkül*[8] bewährt, das ebenfalls auf *Milners* π-*Kalkül* aufbaut.
Das κ-*Kalkül* setzt sich aus sogenannten Lösungen, Agenten und Regeln zusammen. Lösungen sind Multisets von Agenten, welche wiederum die Entitäten der beschriebenen Prozesse repräsentieren. Diese Agenten bestehen aus einem Namen und einem sogenannten *Interface*, welches aus einem Set von Sites besteht. Der Begriff „Site“ ist hierbei äquivalent zu dem aus der Beschreibung von Signalübermittlungen zwischen Proteinen in Kapitel 2.1.
Die Sites eines Agenten besitzen wiederum einen Bindungszustand und einen internen Zustand, wobei letzterer z.B. zur Kodierung der Position in der Zelle oder zur Beschreibung posttranslationaler Modifikationen genutzt wird. Der Bindungszustand beschreibt hingegen, in welcher Beziehung die entsprechende Site zu anderen Sites steht. Jede Site kann dabei entweder gebunden oder ungebunden, also *frei*, sein. Die zuvor erwähnten Lösungen werden nun durch Multimengen von Agenten dargestellt, deren Sites sich in verschiedenen internen und Bindungszuständen befinden. Im weitesten Sinne sind solche Lösungen also Site-Graphen, deren Knoten Agenten und Sites und deren Kanten Verbindungen zwischen Sites sind.
Zu guter Letzt gehört zu jedem Prozess eine Menge an Regeln, welche aus einem Paar einer Lösung S und einer Abbildung der Agenten in S zu konkreten Aktionen bestehen. Diese Aktionen beschreiben nun die verschiedenen Modifikationen von internen Site-Zuständen oder das Erstellen und Auflösen von Verbindungen zwischen verschiedenen Sites, bis hin zur Auflösung von Agenten – in der Biochemie also der Degradation bestimmter Proteine oder Enzyme. Wird die Lösung S nun als Teil eines Systems erkannt, wird die entsprechende Regel aktiviert und die damit verknüpften Aktionen ausgeführt, um eine neue, modifizierte Lösung S' zu erhalten.
In Abbildung 2.3 wird eine solche Regel beispielhaft mit den Agenten *GSK3*β, *p53* und *Mdm2* wie in dem Modell von *Proctor und Gray* in Abschnitt 2.1.1 veranschaulicht. Diese Regel besteht nun aus einer Lösung S, welche all diese Agenten und ihre jeweiligen Konfigurationen beschreibt. In der Lösung S sind alle Sites dieser Agenten frei und besitzen mit Ausnahme der Site `x` von Agent *GSK3*β bzw. Site `t` von Agent *Mdm2* keine internen Zustände. Diese Zustände heißen hier jeweils `u` bzw. `p`. Die Agenten dieser Lösung werden nun auf eine Menge von Aktionen abgebildet, die bei Anwendung der Regel ausgeführt werden sollen. Hier wird *GSK3*β eliminiert und die Agenten *p53* und *Mdm2* über die Sites `a` und `t` verbunden. Hieraus geht nun eine neue Lösung S' hervor, in der *p53* und *Mdm2* verbunden sind und *GSK3*β nicht mehr existiert. Das Zusammenspiel aller solcher Regeln in einem System beschreibt den Prozess nun vollständig. Im Gegensatz zu anderen Prozesskalkülen werden die Vorgänge also über eine sofortige Prüfung der Umgebung von Agenten durch Regeln beschrieben, ohne die konkrete Beschreibung des Erkundungsvorgangs genannter Umgebung.
Die durch die Sprache *Kappa* gegebene Spezifikation muss die Bestandteile dieses Kalküls also vollständig abdecken und spezifizieren können. Da alle Implementierungen dieser Arbeit auf den Gesetzen des κ-Kalküls basieren, ist es elementar, dass auch die hier entwickelte Spezifikation diese Voraussetzung erfüllt.

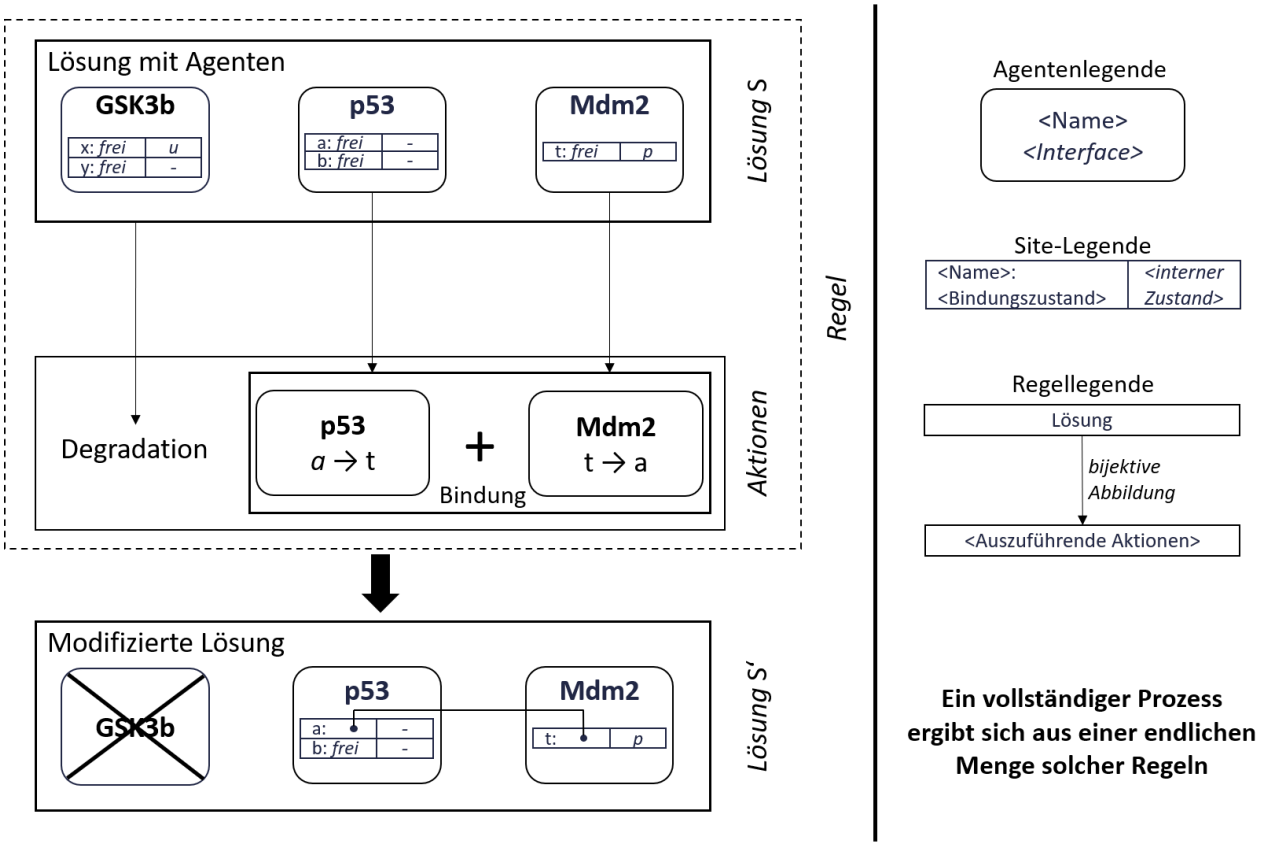


Abbildung 2.3.: Exemplarische Skizze zum κ-*Kalkül*. Es wird eine Bindung von *p53* an *Mdm2* über die Sites *a* und *t* realisiert. Zusätzlich wird ein Agent vom Typ *GSK3*β eliminiert.

2.3. Model-Driven (Software) Engineering

Um bestimmte biochemische Systeme simulieren zu können, sind entsprechende Modelle nötigt. Hierzu versucht das sogenannte *Model-Driven Engineering (MDE)*, abstrakte Repräsentationen für reale Prozesse oder Strukturen zu finden [18]. Ziel ist es hierbei, dass die gewählten Repräsentationen das relevante Themengebiet möglichst präzise abbilden. In dieser Arbeit wird diese Domäne durch biochemische Reaktionen gebildet, wobei beispielsweise die daran beteiligten Moleküle im Modell als Agenten abstrahiert werden.
Weit verbreitet ist Model-Driven Engineering zum Beispiel auch bei der Software-Entwicklung – dann als *Model-Driven Software Engineering (MDSE)* [19]. Hierbei steht es im Vordergrund, ein möglichst präzises Modell für die Funktionalitäten und Programmstruktur der Software zu finden und somit eine fehlerfreie und vielleicht sogar automatisierte Implementierung des spezifizierten Modells gewährleisten.
Zur Spezifikation eines solchen Modells bieten sich domänenspezifische Sprachen an, womit sich Abschnitt 2.3.1 beschäftigen wird. Nachdem deutlich gemacht wurde, wie man Modelle mit domänenspezifischen Sprachen definieren kann, wird in Abschnitt 2.3.2 näher beleuchtet, wie man bestimmte Modelle über Transformationen in andere Modelle überführen kann. Das Ende dieses Kapitels handelt von *Pattern Matching* und warum dies für die Simulation biochemischer Prozesse relevant ist. Im

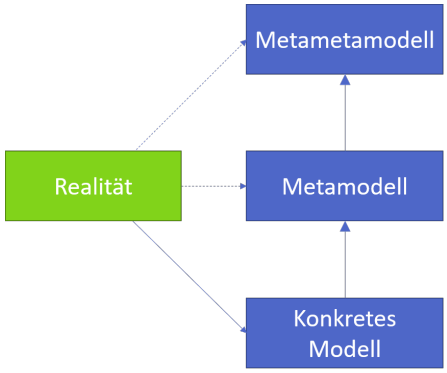


Abbildung 2.4.: Hierarchie der Modellabstraktionen

Model-Driven Engineering stehen - wie der Name schon sagt - Modelle im Mittelpunkt der Entwicklung eines Produkts. Diese Modelle kann man primär kategorisieren in konkrete Modelle, Metamodelle oder sogar Metametamodelle. Metamodelle beschreiben dabei alle möglichen Konfigurationen, die aus diesem Metamodell generierte Modelle annehmen können, und die weiterführende Abstraktionsstufe der Metametamodelle beschreibt alle möglichen Konfigurationen der aus sich erzeugten Metamodelle (s. Abb. 2.4).
Speziell für diese Arbeit und im Kontext der Biochemie sind diese Modelle konkrete Systeme aus Entitäten auf molekularer Ebene, wie in Kapitel 2.1 eingeführt. Biochemische Metamodelle für Simulationen werden aus den entsprechenden Spezifikationen gemäß Abschnitt 2.2.3 gewonnen und beschreiben alle möglichen Agenten innerhalb eines Systems sowie deren Konfigurationen und Dynamiken untereinander. Die hier betrachteten Simulationsmodelle sind allesamt graphenbasiert, da sie eine fundamentale Rolle bei der regelbasierten Simulation von Prozessen spielen, indem sie mithilfe von *Pattern Matching* anhand bestimmter Transformationsregeln transformiert werden.

2.3.1. Domänenspezifische Sprachen

Eine seit Jahren etablierte Methode zur Spezifikation und daraus resultierenden Generierung von Modellen geschieht über domänenspezifische Sprachen („*domain specific language*“, DSL). Konkret über domänenspezifische Modellsprachen, neben den domänenspezifischen Programmiersprachen.
Eine weitere Unterteilung von DSLs ist in interne und externe Sprachen möglich. Die internen DSLs sind dabei Teil einer größeren Elternsprache, während die externen DSLs einen eigenen, unabhängigen Interpreter besitzen.
Das Ziel von domänenspezifischen Sprachen ist es nun im Gegenteil zu general-purpose Sprachen, nur die spezifische Domäne, für die sie entworfen wurden, und diese möglichst präzise und vollständig abzubilden. Die Beschränktheit auf diesen einen Anwendungskontext ist dabei durchaus gewollt und erlaubt eine hochspezifische Darstellung, wo *general-purpose Sprachen*, die mehrere solcher Kontexte abzubilden versuchen, am daraus resultierenden Sprachumfang scheitern. So werden außerdem die Grammatiken von DSLs kompakt gehalten, wodurch die intuitive Verständlichkeit und Usability der Sprache enorm an Wert gewinnen.

Etablierte Sprachen Die beiden weit verbreiteten Biochemie-DSLs *Kappa*[1] und die *BioNetGen-Language*[2] sollen hier eine kurze Einführung erfahren. Um eine konsistente, einfach verständliche Visualisierung der modellierten Zusammenhänge zu erhalten, wird im folgenden eine grafische DSL eingeführt, anhand der die verschiedenen anderen Spezifikationen erklärt werden können.
In dieser grafischen Sprache werden Agenten durch Kreise dargestellt, in denen ihr jeweiliger Name und Typ im Format `<Name>:<Typ>` enthalten ist. Diesen Agenten haften weitere, kleinere Kreise an, die ebenfalls einen Namen besitzen und die Sites des jeweiligen Agenten repräsentieren. Diesen Sites kann wiederum eine Raute anhaften, die den internen Zustand der entsprechenden Site bezeichnet. Verbundene Sites werden durch eine einfache Verbindungskante aneinander gebunden. Eine beispielhafte Regel wie in Abbildung 2.5 ist die Entstehung einer Verbindung zwischen zwei Sites `x` eines Agenten `a` von Typ A und der Site `y` eines Agenten `b` von Typ B, die vorher frei waren, mit einer Zustandsänderung der Site `c` von Agent `a` von Zustand `u` in Zustand `p`.

Abbildung 2.5.: Einfache Verbindung zweier Sites `x` und `y` mit Zustandsänderung der Site `c`

In der Sprache *Kappa* müssen vor der Formulierung von konkreten Regeln zuerst die verschiedenen Agententypen mit all ihren Sites und deren möglichen internen Zuständen in einer sogenannten *Signatur* definiert werden. Wie dies anhand eines Beispiels aussieht, zeigt Listing 2.1. In der Signatur werden über das Schlüsselwort ‘`%agent`’ die Agententypen definiert. Darauffolgend befinden sich in runden Klammern die Sites der jeweiligen Typen, die wiederum in geschweiften Klammern ihre möglichen Zustände enthalten.
Die Beispielregel erhält hier den Beispielnamen `'a.b'` und besteht aus einer linken Seite, die die Vorbedingung definiert, einem darauffolgenden Reaktionspfeil ‘`->`’ und der rechten Seite, die die Nachbedingung definiert. Verbundene Sites werden deklariert, indem sie denselben Index in eckigen

```
/* Signature */
%agent: A(x, c {u p})   # Deklaration eines Agenten A
%agent: B(y)            # Deklaration eines Agenten B

/* Rule */
'a.b' A(x[.], c{u}[.]), B(y[.]) -> A(x[1], c{p}[.]), B(y[1]) #A bindet B
```

Listing 2.1: Bindung von A und B mit Zustandsänderung in Kappa

Klammern enthalten. Freie Sites werden durch einen Punkt '`[.]`' in eckigen Klammern dargestellt. Optional kann Sites in geschweiften Klammern ein interner Zustand zugewiesen werden.

Analog dazu zeigt Listing 2.2 dieses Beispiel in der *BioNetGenLanguage*. Dort fällt auf, dass hier zuerst ein '`model`'-Block als Rahmen für die gesamte Code-Struktur benötigt wird.
Auch hier wird eine Signatur über einen Block namens '`molecule types`' eingeführt. Dort werden alle Agententypen aufgelistet und erhalten wie bei *Kappa* ihre Sites hinter ihrem Namen in Klammern. Mögliche Zustände einer Site werden hier jedoch über den Operator '~' an den Site-Namen angehängt. Die Regeln des Modells werden ebenso in einem eigenen Block namens '`reaction rules`' definiert.

```
begin model
    begin molecule types
        A(x, c~u~p)     // Deklaration eines Agenten A
        B(y)            // Deklaration eines Agenten B
    end molecule types

    begin reaction rules
        a.b: A(x, c~u) + B(y) -> A(x!1, c~p).B(y!1) // A bindet B
    end reaction rules
end model
```

Listing 2.2: Bindung von A und B und Zustandsänderung in BNGL

Die Regel erhält wieder den Namen '`a.b`' und der generelle Regelaufbau ähnelt dem von *Kappa* sehr. Hier werden Verbindungen ebenfalls über Indizes definiert, allerdings nicht in eckigen Klammern hinter den verbunden Sites, sondern mittels des Operators '`!`' und einem zusätzlichen Punkt zwischen den verbundenen Agenten. Ebenso gibt es hier keine Kennzeichnung für freie Sites. Sites, die aufgelistet wurden, werden implizit als frei angenommen.

Zuletzt wird die *Reaction Rules DSL* betrachtet – die bisherige Sprache des Simulationstools *SimSG*, für welches das hier entwickelte Framework entworfen wird. Hierbei wird erneut dasselbe Beispiel genutzt. Die entsprechende Darstellung findet sich in Listing 2.3.
Diese Sprache ist wiederum *Kappa* sehr ähnlich, abgesehen einiger selbsterklärender Abweichungen, wie dem Ersatz des Punktes durch '`free`', um freie Sites zu markieren.

Die hier aufgeführten Sprachen sind sehr umfangreich und bieten selbstverständlich noch viel mehr Modellierungsoptionen, als in diesem einfachen Beispiel gezeigt. Eine ausführliche Einführung in jede Sprache würde den Rahmen dieser Arbeit jedoch sprengen und ist auch gar nicht gewollt. Es soll dem Leser an dieser Stelle lediglich ein Überblick über die Syntax und Möglichkeiten verschiedener domänenspezifischer Sprachen aus dem Biochemiekontext vermitteln.

```
// Signature
agent A(x, c{u, p})
agent B(y)

// Rule
rule a.b {A(c{u}[free]), B(y[free])} -> {A(c{u}[1]), B(y[1])}
```

Listing 2.3: Bindung von A und B und Zustandsänderung in der *Reaction Rules DSL*

2.3.2. Modelltransformationen

Es passiert jedoch nicht selten, dass man Modelle in einer Form vorliegen hat, in der sie erstmal nicht weiter verwertbar sind. Um dem entgegenzuwirken, kann man sie mittels *Modelltransformationen* in die ersehnte Form wandeln.
Diese sind ein wesentlicher Bestandteil des *Modell-Driven Software Engineering* und *Development* und beschäftigen sich mit der Abbildung und Überführung eines Modells in ein anderes Modell. Allgemein lassen sich Modelltransformationen nach Art des Zielmodells, Abstraktionsebene und Direktionalität unterscheiden. Im Zuge dieser Arbeit sind zudem alle betrachteten Modelle graphbasiert.

Modelle können entweder von einem Modell in ein weiteres nicht-textuelles Modell überführt werden (*Model-to-Model*) oder es kann ein Textartefakt als Ergebnis der Transformation vorliegen (*Model-to-Code*) [20]. Des Weiteren kann man Modelle danach unterscheiden, ob sie ein bestehendes Modell im Zuge der Transformation modifizieren oder ob sie ein neues Modell erstellen, dessen Eigenschaften sich aus der Konfiguration des Quellmodells ableiten.
Befinden sich Quellmodell und Zielmodell auf derselben Abstraktionsebene, spricht man von *horizontalen Transformationen*, ansonsten handelt es sich um *vertikale Transformationen*. Eine Transformation, die Abbildungen von Quellmodell zu Zielmodell und auch vice versa beschreibt, wird bidirektional genannt. Befasst sie sich jedoch nur mit der einfachen Transformationsrichtung von Quellmodell zu Zielmodell, ist von einem unidirektionalen Modell die Rede.
Im Allgemeinen werden Modelltransformationen häufig durch eine Menge von Transformationsregeln beschrieben, die jeweils aus einer Vorbedingung (der *Left-Hand-Side*) und einer Nachbedingung (der *Right-Hand-Side* einer Regel) bestehen. Die Vorbedingung beschreibt hierbei Strukturen im Quellmodell vor der Transformation und die Nachbedingung die Strukturen nach der erfolgten Modifikation der Vorbedingung durch die entsprechende Transformationsregel. Die komplette Transformationsdefinition ergibt sich aus der vollständigen Menge aller Transformationsregeln. In graphbasierten Modellen bestehen die Vor- und Nachbedingungen solcher Graphtransformationsregeln konkret aus einer Menge an Knoten und Kanten, welche diese Knoten untereinander verbinden.
Zudem kann man in Bezug auf die Sprache(n), in der das Quell- und Zielmodell formuliert sind, zwischen *endogenen* und *exogenen Transformationen* unterscheiden [21]. Endogene Transformationen werden mit Quell- und Zielmodell innerhalb derselben Sprache vollzogen, dies wäre z.B. das *Refactoring* von Quellcode einer beliebigen Programmiersprache. Bei Exogenen Transformationen erfolgt eine Übersetzung des Quellmodells in einer Quellsprache zu einem Zielmodell, das in einer anderen Sprache formuliert ist. Dies trifft zum Beispiel auf die Generierung von ausführbarem Code (z.B. Bytecode für eine virtuelle Maschine) aus Quellcode zu. Der ausführbare Code und der Quellcode haben dieselbe Semantik, drücken diese jedoch auf verschiedene Weisen aus. Exogene Transformationen sind der Kern des hier entwickelten Frameworks, da die Spezifikation des Nutzers, welche in der entsprechenden DSL formuliert wird, in ein ausführbares Simulationsmodell übersetzt werden muss. Dabei müssen die

Semantiken der biochemischen Reaktionsmodelle inklusive der zugehörigen Simulationsdirektiven erhalten bleibe.
Somit handelt es sich bei den Transformationen in dieser Arbeit also um exogene, horizontale, unidirektionale Model-to-Model-Transformationen.
Um Nutzern bei der Ausführung solcher Transformationen unter die Arme zugreifen, bestehen bereits zahlreiche Tools, Frameworks (z.B. *Viatra*[22] und *eMoflon*[3]) und Sprachen (z.B. die *Atlas Transformation Language, ATL*[23]), welche die Definition und Implementierung der jeweiligen Transformationen erleichtern sollen.

2.3.3. Pattern Matching

Zur genaueren Analyse von Modellen gibt es verschiedene Ansätze. Einer davon ist das sogenannte *Pattern Matching*, um bestimmte – oftmals wiederkehrende – Muster innerhalb bekannter Datenstrukturen bzw. Modelle zu finden.
Es wird zum Beispiel in Verbindung mit regulären Ausdrücken genutzt, um Strings bestimmter Muster zu erkennen. Auch in vielen funktionalen Programmiersprachen spielt es eine große Rolle, indem man den Aufbau von Datenobjekten gegen eine bestimmte Struktur prüft, um so gezielt verschiedene Fälle im Programmablauf zu deklarieren und auf die Attribute des Objekts zuzugreifen.
Ebenso ist eine Anwendung von Pattern Matching insbesondere auf den in dieser Arbeit genutzten graphbasierten Modellen möglich. Das Grundproblem hierbei ist es, ein homomorphes (oder isomorphes) Abbild eines Graphen P – dem *Pattern* – in einem anderen Graphen G zu finden. Dies bezeichnet man als „*subgraph isomorphism problem*“[24] und ist in Abbildung 2.6 visualisiert. Hier werden die Typen der verschiedenen Knoten über ihre Farbe definiert und das Muster P in einem vollständigen Graphen G gesucht. Das rot hervorgehobene, in G gefundene Muster ist ein sogenanntes $Match$ und bildet somit einen Subgraphen in G.

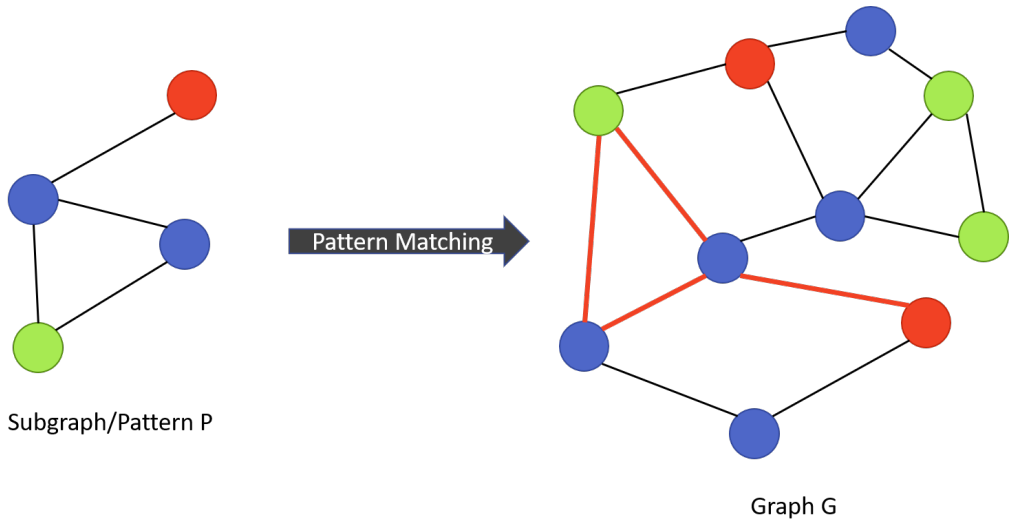


Abbildung 2.6.: Pattern P im Graphen G. Das Match ist in G farblich hervorgehoben.

Da das Finden von Matches in einem Graphen NP-vollständig ist und das jedesmalige neue Ermitteln von Matches bei Transformationen (von Teilen) des Graphen somit zeitlich enorm aufwändig ist, braucht es kostengünstigere Ansätze.

Inkrementelles Pattern Matching Einer dieser Ansätze ist *inkrementelles Graph Pattern Matching*. Hierbei werden bei einer Änderung ΔG des Graphen G nicht von Anfang an alle Matches im um ΔG modifizierten Graphen G' neu berechnet, sondern es werden nur die Änderungen ΔG betrachtet und aufgrund derer ermittelt, ob die bisherigen Matches noch gültig sind bzw. ob durch die Veränderungen neue Matches entstanden sind. Durch diesen inkrementellen Ansatz wird Laufzeit gespart, allerdings zulasten der Speicherauslastung, da alle gefundenen Matches zu jeder Zeit im Speicher gehalten werden müssen.

Rete-Netzwerke Typische Vertreter der inkrementellen Graph Pattern Matcher sind *VIATRA*[22] oder die in *eMoflon* enthaltenen Engines *Democles*[37] und *HiPE*[3]. All diese Pattern Matcher bauen auf sogenannten Rete-Netzwerken auf, die bereits 1987 von Charles Forgy entwickelt wurden [38]. Ein solches Netzwerk besteht aus Knoten, die man in zwei Kategorien unterteilen kann: *selection nodes* und *junction nodes*. Die *selection nodes* definieren Selektionsbedingungen, um einzelne Entitäten in einem Muster zu finden. Im Zuge dieser Arbeit wären das zum Beispiel die verschiedenen Agentenknoten. Die *junction nodes* formulieren Bedingungen und Einschränkungen, um *selection nodes* semantisch zusammenzuführen, indem sie z.B. durch die Kombination zweier Agentenknoten die Verbindung dieser beiden definieren. So könnte eine Bindung von zwei Agenten A und B über die Sites `x` und `c` beispielsweise wie in Abbildung 2.7 aussehen. Hierbei bilden *selection nodes* zunächst die Agenten A und B selbst sowie deren Sites `x` und `c`. Die *junction nodes* definieren weitergehend durch die Zusammenführung der entsprechenden *selection nodes*, dass die Site `x` zu Agent A und die Site `c` zu Agent B gehört. Der letzte *junction node* repräsentiert nun das vollständige Muster aus den bereits zusammengeführten Sites mit ihren zugehörigen Agenten, in dem er diese in sich vereint.

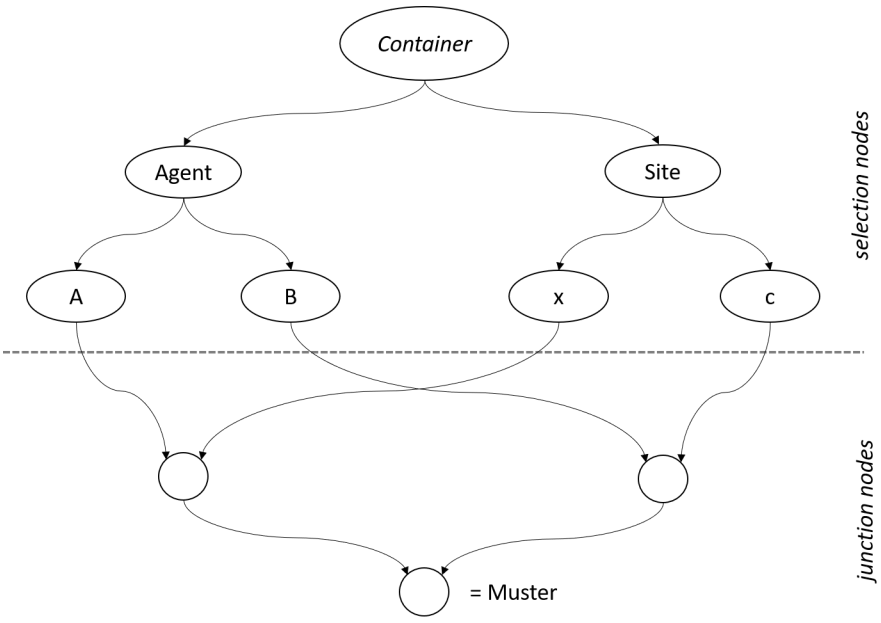


Abbildung 2.7.: Beispielhaftes Rete-Netzwerk des Musters `a.x+b.c`

[3] `github.com/HiPE-DevOps/HiPE-Updatesite`

Während *Democles* und *VIATRA single-threaded* und sequenziell arbeiten, geht *HiPE* einen anderen Weg. Hier war es das Ziel, Pattern Matching auf eine massiv-parallele Ebene zu führen. Dazu wurde Forgys Rete-Konzept mittels eines Aktor-System-Ansatzes[39] auf Basis des *Akka*[4]-Frameworks neu interpretiert, indem die Knoten des Rete-Netzwerks jeweils 1:1 auf eigene Threads abgebildet werden.

Invocations Eine weitere Komponente des Pattern Matchings sind sogenannte *Invocations*, welche die Knoten eines aufrufenden Musterns P auf ein anderes, aufgerufenes Muster I abbilden. Hierbei schränkt die Abbildung eines Knotens aus P auf einen Knoten aus I das aufrufende Muster P weiter ein, da für ein Match von P nun auch der Kontext des aufgerufenen Musters I in der Umgebung des abgebildeten Knotens für ein Match P vorhanden sein muss. Die Einschränkungen aus I gehören hierbei in der Regel nicht zu den jeweiligen Matches von P. Beispielhaft zeigt dies Abbildung 2.8. Das Muster besteht aus einer einfachen Verbindungen zwischen einem blauen und einem grünen Knoten. Dieses Muster taucht zwar zuhauf im Graphen G auf, allerdings wird durch die Abbildung auf das Muster I zusätzlich die Bedingung gestellt, dass der blaue Knoten an einen roten Knoten gebunden ist, wodurch sich die Matches auf die zwei Markierten reduzieren.

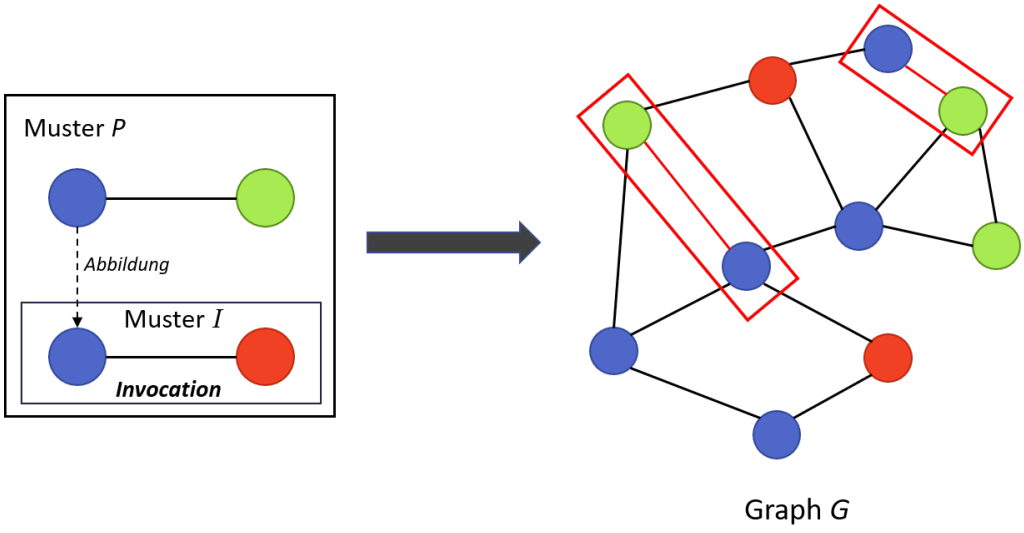


Abbildung 2.8.: *Invocation* des Musters I durch das Muster P. Da der blaue Knoten des Musters P auf den blauen Knoten aus Muster I abgebildet wird, ergeben sich nur die zwei rot hervorgehobenen Matches im Graphen G.

[4] Akka Projektseite: `www.akka.io`

Negative Application Condition Eine *Invocation* lässt sich jedoch auch als sogenannte *Negative Application Condition (NAC)* wie in Abbildung 2.9 nutzen. In diesem Fall wird das aufgerufene Muster I im weitesten Sinne „verboten“, sodass das aufrufende Muster P durch den Pattern Matcher nur noch gefunden wird, wenn der in I abgebildete Kontext **nicht** vorhanden ist.

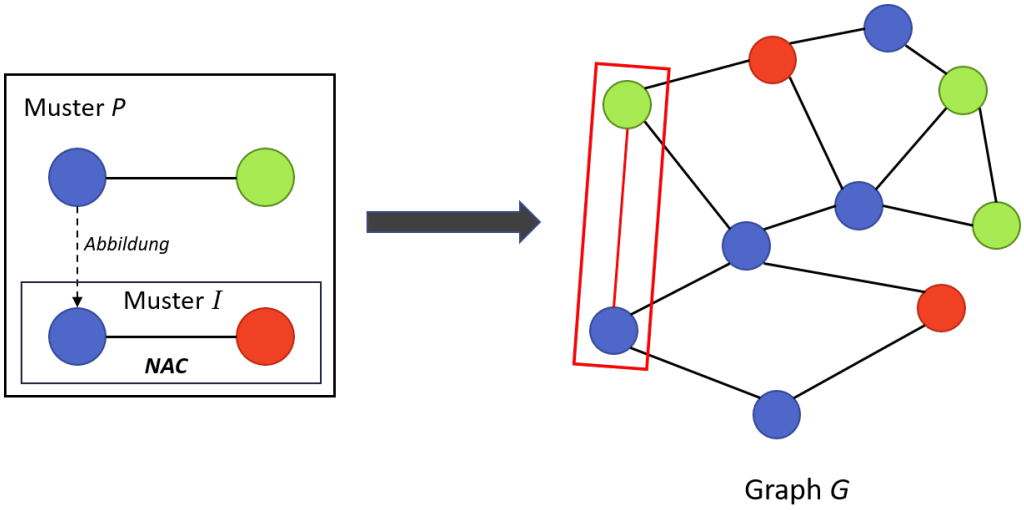


Abbildung 2.9.: *NAC* mit dem Muster I in P. Durch die NAC wird an das Paar aus grünem und blauem Knoten zusätzlich die Bedingung gestellt, dass der blaue Knoten mit keinem roten Knoten verbunden ist. Somit ergibt sich im Graphen G nur ein einziges Match.

Pattern Matching im Biochemiekontext In Modellen biochemischer Systeme werden die verschiedenen Agenten innerhalb des Systems als Knoten in einem Graphen abgebildet und mittels zugehöriger Kanten im Graphen zu größeren Komplexen verknüpft. Die durch entsprechende biochemische Reaktionen eines Modells hervorgerufenen Molekülmodifikationen definieren hierbei implizit die nötigen Graphtransformationsregeln, um den Ablauf einer Reaktion im Graphen abzubilden.
In diesem Kontext ist insbesondere inkrementelles Pattern Matching relevant, da die betrachteten biochemischen Systeme gewöhnlich eine extrem große Anzahl an Molekülen besitzen und somit auch bei entsprechend vielen Reaktionsregeln äußerst viele Matches erzeugen. Bei diesen Systemen wäre es sehr teuer, in jeder Iteration der Simulation alle Matches neu zu berechnen, weswegen inkrementelles Pattern Matching hier eine gute Lösung bietet. Eine Iteration bezeichnet das einmalige Finden aller benötigten Muster in einem Graphen bzw. System und die dazugehörige Anwendung der entsprechenden Transformationsregeln.

3. Implementierung

Die essenziellen Teile der Implementierung des Frameworks bestehen aus der Spezifikation bzw. der Sprache *Re.action* selbst, sowie der Einbindung des aus der Spezifikation resultierenden Modells über weitere Modelltransformationen in das Simulationstool *SimSG*, welches genau wie *eMoflon* in der Programmiersprache *Java* realisiert wurde. Aus diesem Grund wurde die hier entwickelte Implementierung zur einfacheren Integration der *Re.action*-Spezifikation ebenfalls in *Java* umgesetzt. Hierzu beginnt Abschnitt 3.1 mit dem Aufbau und Zusammenhang der verschiedenen Module des Frameworks.
Im darauffolgenden Abschnitt 3.2 werden kurz die zur Modellierung und Realisierung genutzten Tools *Xtext*[1] und das *Eclipse Modeling Framework (EMF)* [33] eingeführt.
In Abschnitt 3.3 wird die domänenspezifische Sprache vorgestellt, indem Grund- und Leitgedanken (3.3.1), sowie technische Details (3.3.2) erläutert werden. Dabei stehen insbesondere Begründungen für bestimmte getroffene und verworfene Designentscheidungen im Vordergrund, aber auch psychologische Aspekte werden berücksichtigt.
Aus der domänenspezifischen Sprache wird ein Spezifikationsmodell gewonnen, das nun in ein passendes Simulationsmodell überführt werden muss. Abschnitt 3.4 widmet sich diesem Transformationsprozess, indem er die verschiedenen Modellarchitekturen und die Transformationen zwischen diesen näher beleuchtet. Dazu zählen das Sprachmodell 3.4.1, ein Zwischenmodell 3.4.2 und das endgültige Simulationsmodell 3.4.3.

Die Implementierung als Ganzes bietet dann eine präzise und intuitive Sprache zur exakten Spezifikation der gewählten Simulationsmodelle, sowie das nötige Framework, um die Simulationen entsprechend mit den gewählten Pattern-Matchern im Simulationstool *SimSG* auszuführen.

3.1. Framework-Module

Das gesamte Framework setzt sich nun aus der Spezifikation und den verschiedenen Modellen, die für die Transformation zum endgültigen Simulationsmodell benötigt werden, zusammen. Somit ergibt sich ein Zusammenspiel verschiedener Module, wie Abbildung 3.1 zeigt.
Das Framework selbst besteht hierbei aus der Spezifikation und einer Menge an Modellen und zugehöriger Transformationen, die die Übersetzung des aus der Spezifikation hervorgehenden Modells in ein ausführbares Simulationsmodell realisieren.
Die Spezifikation besteht aus der Sprache *Re.action* samt Grammatik, Scoping für verschiedene Identifier und Validation, ob die eingegebenen Regeln auch gültig sind und simulierbare Reaktionen darstellen. Hinzu kommt ein Plugin zur Anbindung an die IDE *Eclipse*[2] [25], welche gleichzeitig das *Graphical User Interface* (GUI) zur Eingabe der Spezifikation und Möglichkeiten zur Implementierung des gewünschten Syntax-Highlightings bereitstellt.

[1] `eclipse.org/Xtext/` [2] `eclipse.org/ide/`

Die Modelle des Frameworks sind zum einen das Sprachmodell, welches sich direkt aus der Grammatik der Sprache ergibt, und ein Arbeitsmodell, zu dem das Sprachmodell transformiert wird. Aus diesem Arbeitsmodell wird mit einem letzten Transformationsschritt das vollständige Simulationsmodell für das Tool *SimSG* gewonnen. Dieses besitzt eine sogenannte *Simulation Definition (SimDefinition)* mit genaueren Informationen zu den Regeln und Referenzen zu den für das Pattern Matching benötigten sogenannten *GT-Regeln* und *IBeX-Patterns*, sowie Rahmenbedingungen der Simulation. Ein weiterer Teil des vollständigen Simulationsmodells ist ein Metamodell, das alle möglichen Agententypen und deren Eigenschaften beschreibt.
Das Simulationstool *SimSG* besitzt als Hauptbestandteil eben dieses Simulationsmodell und beruht auf dem Computer-Aided-Software-Engineering(CASE)-Tool *eMoflon*[3], welches die Pattern Matcher *Democles* und *HiPE* zur Verfügung stellt. Die entsprechenden Muster und anzuwendenden Transformationen werden über die *GT-Regeln* und *IBeX-Patterns* für *eMoflon* definiert.

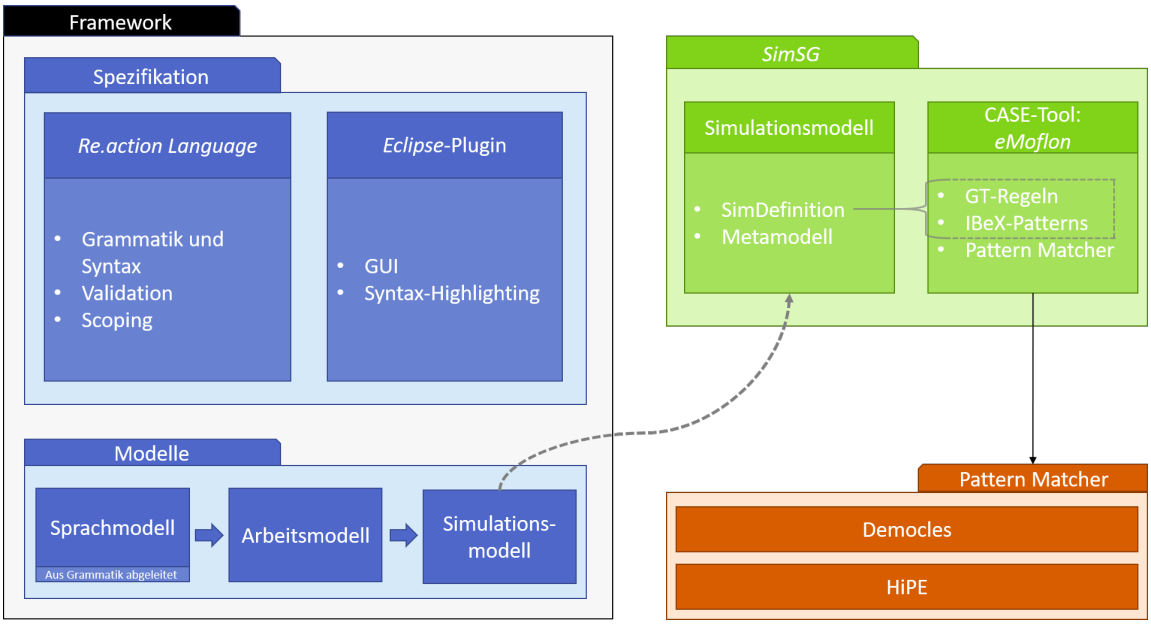


Abbildung 3.1.: Module des Frameworks

3.2. Tools zur Modellierung

Zur Realisierung der in diesem Kapitel dargelegten Implementierung war der Einsatz diverser Tools nötig, um ein korrektes und effizientes Framework zu gewährleisten. Diese Tools werden hier kurz vorgestellt.

3.2.1. *Eclipse Modeling Framework* (EMF)

Ein bewährtes Tool für den Entwurf und die Gestaltung von Metamodellen bietet das als Plug-in für die IDE *Eclipse*[3] verfügbare *Eclipse Modeling Framework*(EMF)[4] [33], mit dem sogenannte *Ecore-Metamodelle* modelliert werden können. Die Beschreibung dieser Metamodelle erfolgt dabei über klassische objektorientierte Ansätze mittels Vererbung und Klassen, die Attribute, Referenzen und Operationen besitzen. Der Vorteil hierbei ist, dass aus dem Metamodell Java-Code generiert werden kann, der aufgrund des objektorientierten Aufbaus des Ecore-Modells direkt äquivalent zum Metamodell ist. Sowohl das Sprachmodell, als auch das Zwischenmodell und die verschiedenen Komponenten des vollendeten Simulationsmodells – namentlich *IBeX-Patterns*, *GT-Regeln* und *Simulationsdefinitionen* – basieren auf *Ecore-Modellen*.
Zudem liefert das Framework eine Fülle an weiteren hilfreichen Klassen und Methoden, um mit den Modellen umzugehen und diese auch weiterführend zu verarbeiten und aufzubereiten. Dies ermöglicht unter anderem nicht nur die statische Erstellung und Generierung von (Meta-)Modellen in *Eclipse*, sondern auch die durch entsprechende Bibliotheken unterstützte dynamische Erstellung von Modellen on-the-fly, was für die hier implementierten Transformationen unerlässlich war.
Dazu gehört außerdem der mögliche Export aktueller Modellinstanzen in *eXtensible Markup Language(XML)*[5] – bzw. das erweiterte *XML Metadata Interchange(XMI)*[6]-Format. Aufgrund der allgemeinen Anwendbarkeit von *XML* auf beliebige hierarchisch strukturierte Daten, ist es für alle möglichen Modelle geeignet. Die einzige Einschränkung hierbei ist, dass alle Daten in einem einzelnen Wurzelobjekt enthalten sein müssen. Der dadurch mögliche einfach Import und Export entsprechender Modellinstanzen war eine große Hilfe bei der Realisierung verschiedener Transformationen im *Re.action*-Framework. Dies ermöglicht es zudem, Zwischenmodelle jederzeit auszugeben und abzuspeichern. Auch das Simulationsmodell erwartet die aktuellen Modellinstanzen im *XMI*-Format und gibt den Zustand des Modells nach Abschluss der Simulation auf dieselbe Weise aus, um mit diesen Ergebnissen beispielsweise die erste Simulation in einer weiteren Simulation mit abgewandelten Regeln oder Rahmenbedingungen fortzuführen.

[3] `eclipse.org/ide/` [4] `eclipse.org/modeling/emf/` [5] `www.w3.org/XML/` [6] `www.omg.org/spec/XMI`

3.2.2. *Xtext*

Zur Erstellung von domänenspezifischen Sprachen gehören bis zum Erreichen voller Funktionalität viele Komponenten und Arbeitsschritte, wie die Implementierung von Interpreter, Parser, Scoping und vielem mehr.
Ein wichtiges Tool zum Entwurf von *Re.action* war *Xtext*[7] [34], das in Verbindung mit dem *Eclipse Modeling Framework* den Entwurf einer DSL direkt in der Entwicklungsumgebung *Eclipse* ermöglicht. Hierbei wird der Aufwand des Sprachentwurfs auf die Definition einer kontextfreien Grammatik über diverse Grammatikregeln basierend auf der *erweiterten Backus-Naur-Form* (EBNF) nach ISO-Standard [35], sowie die Konfiguration des Scopings und weiterer Interpreter-Regeln über eine vorgegebene API reduziert. Die konkreten Datenstrukturen wie der *abstract syntax tree (AST)*, welcher die syntaktische Struktur der Sprachgrammatik repräsentiert, werden automatisch generiert und dem Anwender über einfach nutzbare Schnittstellen zur Verfügung gestellt.
Die in *Xtext* fertig entworfene Sprache steht letztendlich auch als *Ecore*-Modell zur Verfügung, wodurch alle Sprachkomponenten zu entsprechenden Java-Klassen transformiert und so in den weiteren Programmkontext eingebunden werden können.
Durch die Anbindung an *Eclipse* steht Entwicklern und Nutzern direkt eine grafische Nutzeroberfläche zur Eingabe der Sprache zur Verfügung. Diese erkennt automatisch Dateien der entsprechenden DSL und stellt für diese frei konfigurierbares Syntax-Highlighting sowie ein beliebig einstellbares Validationsmodul für Nutzereingaben zur Verfügung. In diesem Modell können programmatisch Bedingungen definiert werden, unter denen der Editor dem Nutzer Warnungen anzeigt oder Fehler bei zum Beispiel ungültigen Regeln wirft, welche die Datei invalidieren.
Auf diese Weise wurde auch in *Re.action* ein umfassendes Set an Warnungen erstellt, welche dem Nutzer die Spezifikation seines Reaktionssystems erleichtern sollen. Dazu gehört unter anderem der Hinweis auf Regeln, die nicht sinnvoll formuliert wurden, da beispielsweise Vorbedingung und Nachbedingung identisch sind. Zudem konnte die Sprache in Hinblick auf ihre Validität um Aspekte erweitert werden, wie z.B. bestimmte Anforderungen an die Formulierung von Regeln, welche durch die reine Grammatik nicht abdeckbar sind, aber auch einfache mathematische Einschränkungen in Verbindung mit einem dafür konstruierten Interpreter für arithmetische Ausdrücke, wie das Verhindern einer Division durch 0, das Ziehen einer negativen Wurzel (Numerische Werte in *Re.action* sind nur reell) oder ähnliches.
Dabei sind die von *Xtext* direkt zur Verfügung gestellten Klassen in dem *Java*-Dialekt *Xtend*[8] verfasst. Projekte in diesem Dialekt sind direkt integrierbar in jedes *Java*-Umfeld, da sie auch zu *Java*-Quellcode kompilieren. Im Kern überführt *Xtend* die Syntax von *Java* in eine kompaktere Form durch zum Beispiel Typinferenz von Objekten. Ansonsten bleibt es *Java* sehr ähnlich, da es primär objektorientiert arbeitet und diese Objekte statisch und stark typisiert sind. Es fügt jedoch auch diverse eigene Features hinzu. So ist es beispielsweise über sogenannte *Template-Ausdrücke* möglich, Vorlagen für Strings zu definieren, was sich für programmatische Code-Generierung als sehr praktisch erweist und beispielsweise die automatische Code- und Modellgenerierung in *Xtext* auf sehr einfache Weise ermöglicht.

[7] `eclipse.org/Xtext/` [8] `www.eclipse.org/xtend/`

3.3. Entwicklungsprozess der *Re.action DSL*

Im Zentrum des Spezifikationsmoduls steht *Re.action* als domänenspezifische Sprache. Design und Entwicklung einer solchen DSL können nach diversen Parametern erfolgen. Der geübte Software-Entwickler wird sich hierbei primär an den ihm bekannten Konventionen aus anderen Programmiersprachen orientieren. Ebenso werden sich die Syntax der Grammatik, sowie die Semantik der verschiedenen Operatoren und Konstrukte, an dem Erfahrungskreis seines Berufsfeldes orientieren. Da es sich hierbei allerdings um eine interdisziplinäre Arbeit handelt, darf nicht vernachlässigt werden, dass Wissenschaftler aus anderen Fachgebieten und Themenfeldern auch andere Assoziationen für bestimmte Konstrukte haben werden. Was ein aus der objektorientierten Programmierung stammender Software-Entwickler für selbstverständlich hält, könnte bei Biochemikern Fragen aufwerfen.

3.3.1. Grundgedanken des Sprachkonzepts

Um eine Sprache zu entwickeln, die für Menschen aus verschiedenen Domänen gleich verständlich ist, ist es wichtig, sich mit den psychologischen und pädagogischen Aspekten von Sprachen auseinanderzusetzen. Eine mögliche Orientierung für das intuitive Sprachverständnis bieten pädagogische Studien zur Erlernbarkeit der entsprechenden Sprachen und psychologische Erkenntnisse zum Sprachverständnis und dem damit verbundenen Assoziationen allgemein. So zeigte sich unter anderem, dass technische Sprachen, die sich an der natürlichen (Alltags-)Sprache mit ganzen Wörtern und Satzteilen bzw. bekannten Assoziationen orientieren, leichter zu erlernen sind, als solche, die abstrakten Operatoren bestimmte Semantiken zuweisen [26] [27] [28]. Das Problem unserer natürlichen Sprache ist jedoch, dass sie viele Binde- und Füllwörter enthält, mit denen eine kompakte Spezifikation nicht möglich ist. Da Kompaktheit jedoch auch ein wichtiger Faktor bei der Spezifikation von Modellen ist, um diese effizient zu erstellen, sollte eine Orientierung an natürlicher Sprache nur bei ausgewählten Schlüsselwörtern erfolgen.
Abstrakte Operatoren sind eine Möglichkeit, um möglichst viel Semantik auf kompakte Art und Weise zu verkapseln. Hier gilt es nun, einen Kompromiss aus Intuitivität und Kompaktheit zu finden, indem man Operatoren wählt, die möglichst viele Nutzer mit derselben Semantik assoziieren, da sie in verschiedenen Themenbereichen mit ähnlichen Bedeutungen auftreten. Zum Beispiel werden die meisten Leute ein Fragezeichen mit Ungewissheit oder Unbestimmtheit verbinden, während eine Raute außerhalb eines Social-Media-Kontexts erstmal keine konkrete, intuitive Assoziation bietet.
Ein weiterer Einflussfaktor, um den Umfang konkreter Formulierungen der Sprache kompakt zu halten, ist das „*don't care, don't write*"-Prinzip. Dieses wurde im Biochemiekontext bereits von Danos et al. [8] in der Sprache Kappa [1] genutzt und basiert darauf, nur die für den aktuellen Kontext wichtigen Informationen spezifizieren zu müssen. Falls zum Beispiel durch den internen Zustand einer Site die aktuelle Lage eines Proteins innerhalb einer Zelle beschrieben wird, dies aber für die formulierte Reaktion keine Rolle spielt, hat man die Option, den internen Zustand der Site einfach gar nicht zu erwähnen. So wird vermieden, Regeln für jede mögliche Lage des Proteins formulieren zu müssen.
Diese Leitlinien bilden das Fundament des hier gewählten Sprachdesigns. Des Weiteren ist es naheliegend, sich an Konventionen zu orientieren, die sich in den relevanten Anwendungsdomänen bereits bewährt haben und etabliert sind. Insbesondere im Biochemiekontext ist es nun naheliegend, sich zur Darstellung von biochemischen Prozessen an der üblichen Schreibweise von Reaktionsgleichungen wie zum Beispiel der Knallgasgleichung (2.1) zur Entstehung von flüssigem Wasser aus gasförmigem Wasser- und Sauerstoff zu orientieren. Eine solche Gleichung besteht aus einer Menge an Molekülen vor Ablauf der Reaktion (die sogenannten Reaktanten), einem Reaktionspfeil als Reaktionsoperator

und einer Menge an Molekülen nach der Reaktion (den sogenannten Produkten). Ein essentielles Ziel des Sprachdesigns wird es sein, die Spezifikation von Reaktions- bzw. Transformationsregeln möglichst nah bei der Syntax solcher Reaktionsgleichungen zu belassen. Dieser Gedanke liegt allen in Abschnitt 2.3.1 vorgestellten Sprachen – *Kappa*, *BNGL*, und *SimSGL* – zugrunde. Im Zuge der genauen Darlegung der hier implementierten Spezifikation *Re.action* wird auffallen, dass einige Features an den bereits etablierten Sprachen angelehnt ist. Dies ist durchaus so gewollt, da das Ziel von *Re.action* ist, eine Mischung der besten Features dieser Sprachen in sich zu vereinen und sinnvoll zu erweitern.
Ein weiterer wichtiger Punkt zur allgemeinen Bedienbarkeit der Sprache ist der Aufwand, mit dem die Eingabe von Formulierungen in der Sprache verbunden ist. Im Allgemeinen ist es unpraktisch, große Mengen an ungewöhnlichen beziehungsweise selten genutzten Sonderzeichen zu haben. Auch wenn diese Sonderzeichen oft Symbole sind, die großen semantischen Wert in sich tragen, und zum Beispiel Klammern zur Strukturierung des Dokuments beitragen, führen sie in großen Mengen meist zu unübersichtlichem Text. Zudem benötigt die Eingabe von Sonderzeichen auf den meisten Tastaturen komplexere Tastenkombinationen als einen einfachen Tastendruck, was die Texterstellung tendenziell unangenehmer gestaltet.

3.3.2. Features und Syntax der Sprache

Neben des generellen Designs zur Ausgestaltung der verschiedenen Sprachfeatures in Bezug auf unter anderem Verständlichkeit und Kompaktheit sollte das primäre Ziel einer domänenspezifischen Sprache sein, die betrachtete Anwendungsdomäne vollständig abzubilden. Sie muss also alle möglichen auftretenden Entitäten und deren mögliche Zustände sowie Konfigurationsmöglichkeiten spezifizieren können. Speziell im Kontext der hier zu modellierenden regelbasierten biochemischen Reaktionen, existieren pro Reaktion folgende Entitäten:

- **Agenten** sind die Kernelemente jeder Reaktionen. Sie abstrahieren das Konzept der biochemischen Moleküle, die als Reaktanten bzw. Produkte in der Vor- bzw. Nachbedingung einer Reaktionsregel auftreten können.
- **Sites** sind Subkomponenten von Agenten und stellen die Schnittstellen für Verbindungen zwischen Agenten dar. Besitzt ein Agent keine Sites, kann er auch keine Verbindungen eingehen.
- **Interne Zustände** sind weitere spezifizierbare Eigenschaften von Sites. Jede Site kann sich in einem internen Zustand befinden, der weitere Informationen wie zum Beispiel die aktuellen topologischen Eigenschaften des Moleküls oder die momentane Lage innerhalb einer Zelle repräsentiert. Der interne Zustand einer Site ist nicht zu verwechseln mit ihrem Bindungszustand!

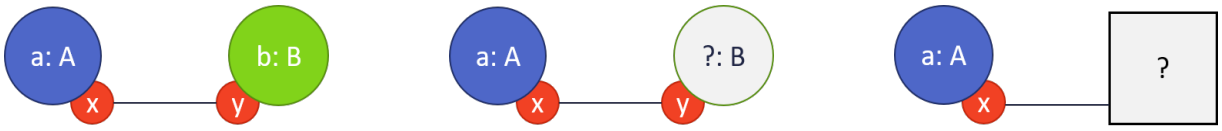
Da es natürlich nicht ausreicht, die verschiedenen Entitäten eines Modells zu beschreiben, ohne auf ihre Beziehungen untereinander einzugehen, müssen auch die verschiedenen möglichen Bindungsverhältnisse der Agenten unter- und miteinander beachtet werden. Hierbei ist es von Vorteil, die naive Überlegung von Sites, die nur gebunden oder ungebunden sein können, beiseite zu legen. Vielmehr spielt es nämlich auch eine Rolle, ob und wie genau der Bindungspartner gebundener Sites bekannt ist. Dazu wird die Klassifikation von Bindungsstatus des *κ-Kalküls* aufgegriffen, woraus sich folgende Menge an möglichen Bindungsdefinitionen ergibt [8]:

- **Freie Sites** sind genau das, was der Name schon andeutet. Ihr Bindungszustand ist „*frei*“ und somit sind sie ungebunden. Sollten alle Sites einer Agenteninstanz frei sein, heißt diese Instanz insgesamt „*frei*“.

- **Gebundene Sites** sind mit einer anderen Site des eigenen „Elternagenten“ oder eines fremden Agenten verbunden. Nun ist aber noch offen, um was für einen Agenten es sich beim Bindungspartner handelt:

  i. Einen konkreten Agenten, mit dem die Site verbunden ist (3.2a)

  ii. Einen beliebigen Agenten, an den die Site gebunden ist, ohne dass bekannt ist, um welchen Agenten es sich genau handelt. In diesem Fall spricht man von einem *unterspezifizierten* Bindungszustand. Hierbei kann man noch danach unterscheiden, ob zumindest der Typ des verbundenen Agenten (z.B. ein bestimmtes Protein) bekannt ist (3.2b), oder auch dieser gänzlich unbekannt ist. (3.2c). Bei Angabe eines Agententyps in unterspezifizierten Bindungen ist auch die Site anzugeben, an die am unbekannten Partneragenten gebunden wird.



(a) Konkret (b) Unterspezifiziert mit Typ (c) Unterspezifizert ohne Typ

Abbildung 3.2.: Visualisierung der verschiedenen Bindungsarten anhand eines Agenten *a* von Typ A

- **Unspezifizierte Sites** enthalten keine Information darüber, ob sie gebunden oder frei sind. Dies ist insbesondere für das „*don't care, don't write*“-Prinzip sinnvoll, falls der genaue Bindungszustand eines Agenten keine Rolle spielt.

Hierbei gilt es zu beachten, dass der interne Zustand einer Site immer unabhängig vom Bindungszustand der Site ist.
Abgesehen von diesen Modellierungsoptionen, die die Spezifikation für die reine Darstellung der biochemischen Prozesse bieten muss, sind natürlich weitere Komponenten vonnöten, um das Simulationsmodell zu vervollständigen. Diese Komponenten werden im Folgenden an Beispielauszügen der Sprache erläutert. Hierbei werden die verschiedenen Sprachfeatures exemplarisch und auch visuell anhand der grafischen DSL aus Abschnitt 2.3.1 bzw. Abbildung 2.5 präsentiert. Das Metamodell der hier genutzten Beispiele besteht aus denselben Agententypen: Einem Agententyp A mit Sites `x` und `c`, wobei sich Site `c` in den Zuständen `u` und `p` befinden kann, und einem Agententyp B mit Site `y` ohne weitere interne Zustände.

Modellsignatur

Das gesamte Modell besitzt eine eigene Signatur, in der die Agenten mit ihrem Namen, ihren möglichen Sites und deren möglichen internen Zuständen festgelegt werden. Die Definition von Agenten ist zwar theoretisch in jeder Zeile des Quellcodes ad hoc möglich, ohne dass die Definition selbst zu einer formalen Signatur als alleinstehendem Block gehört – aus Gründen der Übersichtlichkeit und Einheitlichkeit sollten jedoch alle Agentendefinitionen gemeinsam zu Beginn des Modells aufgeführt werden. Jede Instanz eines Agententyps besitzt alle hier definierten Sites, die sich immer in einem konkreten internen Zustand befinden, insofern überhaupt interne Zustände für die entsprechende Site definiert wurden.
Wie die Definition von Agententypen in der hier entwickelten Spezifikation *Re.action* erfolgt, zeigt

Listing 3.1. Hier werden Agenten gemäß Abbildung 3.3 definiert. Die Syntax ist hierbei quasi analog zu *Kappa* mit leicht variierter Klammerung und Zeichensetzung. Kommentare werden in *Re.action* mittels einer vorangestellten Raute '#' für die jeweilige Zeile eingeleitet und erhalten vom Interpreter der Sprache keine weitere Beachtung.

```
1  agent A: x, c(u, p)  #Definition Type A
2  agent B: y           #Definition Type B
```

Listing 3.1: Definition von Agententypen in *Re.action*

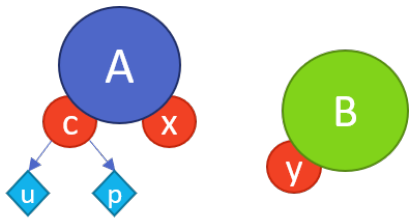


Abbildung 3.3.: Agententypen A und B

Regeln

Der essenzielle Teil des Prozessmodells besteht selbstverständlich aus den entsprechenden Reaktions- beziehungsweise Transformationsregeln. Zur Erklärung der Struktur und Syntax verschiedener Konstrukte wird hier eine weitere Sprache eingeführt. Sie schreibt alle (Sonder-)Zeichen und Schlüsselwörter der Konstrukte aus und kennzeichnet die mit konkreten Ausdrücken der DSL zu ersetzenden Komponenten mit der in '<, >' gehüllten Bezeichnung der entsprechenden Komponente. Sie orientiert sich im weitesten Sinne an der Backus-Naur-Form(BNF) [29]. Die komplette DSL in Backus-Naur-Form findet sich in Appendix A. Transformationsregeln haben im Allgemeinen die Form aus Listing 3.2. In

```
rule <Name> (<Signatur>):   <Vorbedingung> => <Nachbedingung>   @<Reaktionsrate>
```

Listing 3.2: Allgemeine Form von Regeln in *Re.action*

anderen Sprachen wie *Kappa* werden die einzelnen Agenteninstanzen nur über die Reihenfolge, in der die Instanzen innerhalb der Vor- und Nachbedingungen aufgeführt werden, zugeordnet und auf Namen wird gänzlich verzichtet. Da dies sehr fehleranfällig und schwer nachvollziehbar ist, gibt die Reaktionssignatur in *Re.action* namentlich die Instanzen aller Agenten an, die an der Reaktion beteiligt sind, um dieses Problem zu lösen. Eine vollständige Reaktionssignatur besitzt also sowohl die Namen der jeweiligen Agenteninstanzen als auch die zugehörigen Agententypen.
Die Vorbedingung besteht wie z.B. in Kappa auch aus den entsprechend konfigurierten Agenten vor der Transformation. Die Nachbedingung gibt an, welche Agenten in welcher Konfiguration nach der erfolgreichen Transformation bzw. nach Ablauf der biochemischen Reaktion vorliegen. Allgemein ist bei Vor- und Nachbedingung auch von Mustern oder Pattern die Rede, da sie auf Graphenebene für Pattern-Matcher schließlich nichts anderes darstellen. Zwischen Vor- und Nachbedingung befindet sich ein Reaktionspfeil '=>', der die beiden Bedingungen wie schon in der etablierten, konventionellen Schreibweise chemischer Reaktionen miteinander verbindet (vgl. Gleichung 2.1).
Am Ende der Regel folgt eine Reaktionsrate. Diese Rate ist nach Gillespie [12] die *stochastische Reaktionskonstante* und einheitenlos. Gillespie selbst beschreibt diese eher als eine „*Reaktionswahrscheinlichkeit pro Zeiteinheit*"[12]. Es können also Werte auf Basis einer beliebigen Zeiteinheit eingesetzt werden, allerdings sollte die gewählte Einheit über das Modell hinweg konsistent bleiben. Diese Reaktionsraten sind während der Erläuterung der Modellierungsoptionen verschiedener Regeln der Einfachheit halber

alle auf den Wert 1.0 gesetzt, können jedoch theoretisch beliebige positive reelle Zahlen als Wert annehmen. Wie solche Regeln in *Re.action* konkret aussehen, zeigt Listing 3.3.

```
1  rule simpleBinding  (a: A, b: B):   a.x//0, b.y//0  => a.x+b.y  @1.0
2  rule simpleSplit    (a: A, b: B):   a.x+b.y          => a.x//b.y @1.0
3  rule stateChange    (a: A):         a.c(u)           => a.c(p)   @1.0
4  rule synthesis      (a: A):         _                => a        @1.0
5  rule degradation    (a: A):         a                => _        @1.0
```

Listing 3.3: Einfache Beispielspezifikationen aus *Re.action*

Bindungen erstellen Die Regel in Zeile 1 trägt den Namen *simpleBinding* und stellt genau das dar. Hier wird nun auch die genaue Syntax der Reaktionssignatur deutlich. Es werden alle in der Reaktion involvierten Agenteninstanzen durch Kommata getrennt mit ihrem Namen aufgeführt und anschließend mit einem Doppelpunkt `:` ihr jeweiliger Agententyp definiert. Die allgemeine Form ist also `<Instanzname>:<Typ>`.
In der Vorbedingung der Regel wird über den Ausdruck `a.x//0` auf die Site `x` der Instanz `a` zugegriffen. Der Zugriffsoperator '`.`' für die Sites eines Agenten ist der Domäne der objektorientierten Programmierung entnommen, wo der Operator üblicherweise für den Zugriff auf Datenfelder oder Funktionen eines Objektes genutzt wird.
Der Operator '`//`' ist zweiwertig und drückt in einer Vorbedingung aus, dass zwei Entitäten nicht miteinander verbunden sein sollen, bzw. in einer Nachbedingung, dass die Verbindung zwischen den zwei gegebenen Entitäten gespalten wird. Als Symbole wurden zwei Schrägstriche gewählt, da sie diese Spaltung visuell gut veranschaulichen. Der feststehende Ausdruck '`//0`' drückt hierbei aus, dass die Site auf der linken Seite explizit mit nichts verbunden, also frei ist. Dieses abstrakte „nichts", das in vielen gängigen Programmiersprachen durch die Schlüsselwörter `void` oder `null` dargestellt wird, erhält hier das Symbol der Ziffer `0`. Die Null wird intuitiv oft mit einer Abwesenheit oder einem Mangel von etwas assoziiert und bildet somit die passende Wahl.
Demnach formuliert die Vorbedingung der ersten Regel einen Agenten A mit freier Site x und einen Agenten B mit freier Site y. Wenn man sich auf das Metamodell mit der Definition der Agententypen aus Abbildung 3.3 zurückbesinnt, fällt auf, dass die Site `c` von Agent A und deren interner Zustand keinerlei Erwähnung finden. Dies lässt sich auf das in Abschnitt 3.3.1 eingeführte Konzept „*don't care, don't write*" zurückführen. Was nicht von Belang für die formulierte Reaktion ist, wird nicht modelliert. Für die Regel ist es also egal, ob die Site `c` verbunden ist oder nicht und es spielt auch keine Rolle, ob sie sich im internen Zustand `u` oder `p` befindet. Solche Eigenschaften heißen unspezifiziert.
In der Nachbedingung finden sich dieselben Sites wie auch schon in der Vorbedingung der Regel wieder, allerdings diesmal nicht jeweils als *frei* gekennzeichnet, sondern mittels des Operators '`+`' verbunden. Dieser Operator zur Indikation von Verbindungen hat ebenfalls insofern einen Sinn, dass er die bereits vorhandene Bedeutung des Plus als verbindendes oder zusammenzählendes Elements ausnutzt.
So modelliert die erste Regel einfach die in Abbildung 3.4 dargestellte Transformation.

Abbildung 3.4.: Visualisierung der Regel `simpleBinding`

Bindungen auflösen Die Regel in Zeile 2 namens `simpleSplit` bleibt ebenfalls ihrem Namen treu und stellt die Aufspaltung einer bereits vorhandenen Verbindung dar. Der Einfachheit halber wird von derselben Bindung wie in der ersten Regel ausgegangen.
Die Vorbedingung besteht also hier aus der Verbindung zwischen den Sites `x` und `y`. Interner und Bindungszustand der Site c spielen hier erneut keine Rolle. In der Nachbedingung wird nun nochmal die spaltende Semantik des Operators '`//`' deutlich.
Da die zweite Regel tatsächlich nur den genau umgekehrten Ablauf der ersten Regel beschreibt, bietet dies die Möglichkeit, die beiden Regeln zu einer einzigen bidirektionalen Regeln zusammenzufassen. Wie dies aussieht, stellt Listing 3.4 dar. Es fällt auf, dass sie fast identisch zur Regel `simpleBinding`

```
rule bidirectional(a: A, b: B): a.x//0, b.y//0 <=> a.x+b.y  @1.0,1.0
```

Listing 3.4: Bidirektionale Regel

ist. Die einzigen Unterschiede sind die Änderung des unidirektionalen Reaktionspfeiles '`=>`' zum bidirektionalen Reaktionspfeil '`<=>`' und die Angabe einer weiteren, zweiten Reaktionsrate. Dies ist unabdingbar, da der Prozess in Rückrichtung sonst keine definierte Reaktionsrate hätte. Die Syntax der Reaktionsraten für bidirektionale Regeln ist also wie folgt: `@<RateVorwärts>, <RateRückwärts>`. Hierbei gilt es jedoch zu beachten, dass die Schreibweise der Vorbedingung zwingend in der Form wie in Listing 3.4 erfolgen muss. Dies rührt daher, dass eine Darstellung über `a.x//b.y` wie in der Nachbedingung von Regel `simpleSplit` als reine Nachbedingung zwar ausreichend ist, als Vorbedingung allerdings nur bedeuten würde, dass Site `x` nicht mit Site `y` verbunden ist, aber eventuell trotzdem mit einer beliebigen anderen Sites (hier z.B. mit der agenteneigenen Site `c`). Diese Darstellung bedeutet **nicht** implizit, dass die Site auch *frei* ist. Dies wäre nur der Fall, falls sich aus der Menge aller Regeln innerhalb eines Modells ableiten lässt, dass die Site, mit der die Verbindung verboten wurde, auch die einzige Site ist, an die die Site der linken Seite des Operators jemals bindet.

Zustandsänderungen Die dritte Regel aus Listing 3.3 zeigt den einfachen Wechsel des internen Zustands einer Site. Diesmal wird der Bindungszustand aller Sites vernachlässigt. Speziell bleibt der Bindungszustand der Site `c` unspezifiziert, indem die Agenteninstanz, die Site und ihr interner Zustand zwar erwähnt, aber nicht über einen der beiden Bindungsoperator '`+`' oder '`//`' verknüpft werden.
In solchen unspezifizierten Fällen ist von enormer Wichtigkeit, dass die genaue Definition einer unspezifizierten Eigenschaft in der Nachbedingung verboten ist. Unspezifiziertheit bedeutet z.B. in Bezug zu Bindungszuständen nichts anderes, als dass die betroffene Site sowohl verbunden als auch frei sein kann. Gibt man die entsprechende Site nun rechts als verbunden an, ist die Transformationsregel in dieser Form nicht eindeutig definiert, da unklar ist, ob sie nur aus der Entstehung der neuen Verbindung oder auch der Auflösung einer eventuell vorher existenten Verbindung besteht.
Analog lässt sich dieses Beispiel auf andere Bindungszustände oder interne Zustände übertragen. Im Zuge der Arbeit werden noch weitere solcher Fälle aufkommen. In der Nachbedingung einer Regel müssen alle *erwähnten* Bindungs- und internen Zustände eindeutig spezifiziert sein, insofern sie in der Vorbedingung der Regel nicht schon auf dieselbe Weise unspezifiziert sind wie in der Nachbedingung.

Synthese und Degradation Die vierte und fünfte Regel aus Listing 3.3 stellen einen weiteren neuen Operator vor: '`_`', der im weitesten Sinne nur ein Platzhalter ist. Eine solche „Platzhalter"-Funktion nimmt das *Underscore*-Symbol auch bereits in Pattern-Matching-Konstrukten anderer etablierter Programmiersprachen, wie beispielsweise *Scala*[30] oder *Haskell*[31] ein. Er repräsentiert an Stelle von

Vor- oder Nachbedingung das „leere Muster“, also die Abwesenheit jeglicher Agenteninstanzen.
Die Symbolik des Operators soll für eine möglichst freie bzw. leere Fläche stehen. Dies wäre am leichtesten durch einen Punkt ‘.’ zu bewerkstelligen, da dieser jedoch schon als Zugriffsoperator auf die Sites eines Agenten genutzt wird und Mehrfachassoziationen auf dieselben Komponenten der Sprache für Klarheit und Eindeutigkeit vermieden werden, ist dies die beste Wahl.
Die Notwendigkeit eines solchen Operators wird direkt in Regel 4 bzw. 5 deutlich. Sie stellen die komplett neue Entstehung/Synthese bzw. Auflösung/Zersetzung von Molekülen dar. Dazu wird für die Entstehung neuer Agenten die Vorbedingung mit ‘_’ markiert, da die neuen Agenten ja quasi aus dem „nichts“ heraus entstehen, und für die Auflösung von Agenten wird die Nachbedingung mit ‘_’ markiert, da nach erfolgter Transformation „nichts“ mehr übrig ist. Im Allgemeinen erfolgt die Löschung bzw. Entstehung eines Agenten, indem er nur in der Vor- oder Nachbedingung der Regel Erwähnung findet und auf der jeweils anderen Seite der Regel gar nicht aufgetaucht. Ergo können Entstehungen und Löschungen auch innerhalb einer Regel modelliert werden, die andere Agenten enthält, welche nicht gelöscht oder neu erstellt werden, sondern von der Vorbedingung ausgehend auch in der Nachbedingung fortbestehen.
Es fällt auf, dass die zuvor bei Regel ‘`stateChange`’ definierten Einschränkungen für unspezifizierte Sites in Nachbedingungen durch die vierte Regel verletzt werden. Dort findet sich eine Agenteninstanz `a`, deren Sites keine definierten Bindungs- und internen Zustände haben, während sie in der Vorbedingung gar nicht auftaucht. Bei der Entstehung von Agenten wird dies als Sonderfall zur Kompaktisierung der Schreibweise betrachtet. Instanzen, die auf solche Weise neu entstehen, werden dabei automatisch bei der Übersetzung konkretisiert, indem ihre Sites in einen freien Bindungszustand und die internen Zustände in einen Standardzustand versetzt werden. Dieser Standardzustand wird bei der Definition der Agenten in der Modellsignatur festgelegt. Der erste in der Modellsignatur definierte Zustand einer Site ist stets ihr Standardzustand.

```
1  rule underspecified (a: A, b: B):   a.c+?        => a.c+?, b      @1.0
2  rule underspecType  (a: A):         a.c(u)+B.y   => a.c(p)//0     @1.0
```

Listing 3.5: „Unterspezifizierte“ Bindungszustände

Listing 3.5 führt nun eine neue Art von Bindungstypen ein, sogenannte *unterspezifizierte* Bindungen. Die Regel in Zeile 1 führt dazu den Operator ‘?’ als sogenannte *Wildcard* ein. Diese Wildcard symbolisiert einen beliebigen Agenten, dessen Typ und Konfiguration unbekannt sind. Hierbei wird ebenfalls wieder die bereits intuitiv assoziierte Semantik des Fragezeichens ausgenutzt, das zumeist für etwas Unbekanntes oder Unklares steht. Genau dieser Fall eines „unbekannten“ Agenten tritt hier nun auf. In dem Muster ist also modelliert, dass die Site `c` zwar verbunden ist, allerdings nicht, mit welchem Agenten genau. Die Site ist somit unterspezifiziert. Die Regel ist in Abbildung 3.5 visualisiert.
Auch hier gilt Ähnliches wie bereits bei unspezifizierten internen Zuständen von Sites: Sobald der Bindungszustand einer Site in der Vorbedingung als un- oder unterspezifiziert mit unbekanntem Typ deklariert ist, kann er in der Nachbedingung nicht spezifiziert werden, da die Transformation sonst

Abbildung 3.5.: Visualisierung der Regel `underspecified`

nicht eindeutig definiert ist. Es kann nämlich nicht festgestellt werden, um was für einen Knoten es sich konkret handelt, solange der Typ dieses Knotens nicht gegeben ist. Diese Einschränkung ist bewusst so gewählt worden und gewährt eine starke Typisierung der Knoten im späteren Simulationsmodell. Anhand dieser Regel wird ebenfalls deutlich, wie neue Agenten entstehen können ohne eine Entstehungsregel mit '_' in der Vorbedingung definieren zu müssen, sodass weitere über die Transformation hinweg bestehende Instanzen in der Regel definiert werden können. Auf der rechten Seite entsteht die neue Agenteninstanz `b`, deren Typ in der Signatur als Typ B definiert ist. Die Löschung von Agenten unter Aufführung weiterer fortbestehender Agenten wird genau umgekehrt durchgeführt.

Die zweite Regel in Listing 3.5 befindet sich auf einem Abstraktionsniveau zwischen unterspezifizierten Bindungen und konkreten Bindungen. Die Site `c` ist wieder unterspezifiziert, allerdings wird diesmal zusätzliche Information zum Bindungspartner bereitgestellt, der somit nicht gänzlich unbekannt ist. Es wird nicht einfach die Wildcard '?' genutzt, sondern ein Agententyp an die Stelle gesetzt, wo normalerweise die zugehörige Agenteninstanz der Site aufgeführt wird, an die gebunden wird. Es muss sogar eine konkrete Site dieser unbekannten Agenteninstanz genannt werden, an die gebunden werden soll. Lediglich um welche Agenteninstanz es sich genau handelt, ist nicht spezifiziert (siehe Abbildung 3.6). Hier gelten **nicht** die Einschränkungen für Unspezifiziertheiten wie bei der Regel in Zeile 1. Hier kann die Verbindung auch mit der unbekannten Agenteninstanz aufgelöst werden, obwohl sie nicht konkret bekannt ist, da zumindest ihr Typ gegeben ist.
Da die unbekannte Instanz vom Typ B nicht in der Signatur aufgeführt ist, wird sie auch in der Nachbedingung nicht gelöscht, obwohl sie dort nicht erneut auftaucht. Das wäre ohnehin nur erlaubt, wenn sie nochmals an `a.c` bindet und unter der Annahme, dass in diesem Fall die unterspezifizierte Bindung aus der Vorbedingung erhalten bleibt, da die Nachbedingung in Bezug auf die konkrete Instanz, die durch Aufruf über den Agententyp B beschrieben werden soll, sonst äquivok wäre.

Abbildung 3.6.: Visualisierung der Regel `underspecType`

Initialisierungen

Nun ist ein Modell, das nur aus Regeln besteht, allerdings immer noch nicht vollständig. Denn damit Transformationsregeln angewandt werden können, muss erstmal eine bestimmte Grundmenge an Agenten bzw. Molekülen vorhanden sein. Dazu werden Anfangsbedingungen benötigt, die den Ausgangszustand des Systems vor Anwendung jeglicher Regeln definieren.
Diese Anfangsbedingungen werden in *Re.action* über sogenannte `init`-Konstrukte oder auch Initialisierungen formuliert. Initialisierungen benötigen wie Regeln auch eine eigene Signatur. So wie in Regeln die Muster für Vor- und Nachbedingungen definiert werden, wird in einer Initialisierung nun ein Muster als Anfangsbedingung angegeben, aufgrund dessen eine bestimmte Menge von Agenteninstanzen in einer definierten Konfiguration erzeugt werden soll. Die allgemeine Form von Initialisierungen zeigt Listing 3.6. Bei der Formulierung des Musters gelten hierbei dieselben Regeln wie auch bei

```
init <Anzahl> (<Signatur>): <Anfangsbedingung>
```

Listing 3.6: Struktur zur Definition von Anfangsbedingungen in *Re.action*

Nachbedingungen von Regeln: Es sind keine unter- oder unspezifizierten Modellierungen erlaubt. Denn es ist natürlich nicht möglich, Agenten zu erzeugen, deren Sites zum Beispiel keinen definierten Bindungszustand haben. Alle Sites müssen explizit gebunden oder ungebunden sein und – insofern sie über interne Zustände verfügen – einen eindeutigen internen Zustand besitzen.
Die Anzahl der zu erzeugenden Instanzen muss hierbei aus dem natürlichen Zahlenbereich stammen. Um trotzdem die Möglichkeit zu erhalten, Agenteninstanzen möglichst kompakt zu spezifizieren, kommen dieselben Annahmen wie in Regel 4 aus Listing 3.3 zur Anwendung. Undefinierte Sites werden implizit als *frei* angenommen und undefinierte interne Zustände automatisch auf ihren Standardzustand (der erste für die Site definierte interne Zustand in der Modellsignatur) gesetzt.
Die Erzeugung von 100 beispielhaften Instanzen `a` und `b`, deren Sites alle ungebunden sind und deren interne Zustände sich alle in ihrem Standardzustand befinden (hier nur Site `c` in Zustand `u`) zeigt Listing 3.7.

```
1 init 100 (a: A, b: B): a.x//0, a.c(u)//0, b.y//0    # Standardform
2 init 100 (a: A, b: B): a, b                         # Kurzform
```

Listing 3.7: Initialisierung von 100 freien Agenten `a` und `b` mit allen Sites im Standardzustand

Variablen

Mit Reaktionsraten von Regeln und Initialisierungsanzahlen von Initialisierungen wurden bereits alle in *Re.action* enthaltenen Komponenten eingeführt, in denen numerische Werte vorkommen können. Aus Gründen der Übersichtlichkeit, Fehlervermeidung und um leicht Änderungen am Modell vornehmen zu können, ist es von Vorteil, an diesen Stellen Variablen einsetzen zu können. Die Definition einer

```
var <Name> = <ArithmetischerAusdruck>
```

Listing 3.8: Struktur einer Variablendefinition in *Re.action*

Variable erfolgt über das Schlüsselwort ‘var’ und obliegt der Struktur in Listing 3.8. Variablen können an jeder Stelle im Quellcode definiert werden und mittels ihres Namens im Regelfall global überall dort referenziert werden, wo ein numerischer Wert erwartet wird.
Hierzu stellt *Re.action* die Auswertung folgender arithmetischer Ausdrücke zur Verfügung: Addition(‘+’), Subtraktion(‘-’), Multiplikation(‘*’), Division(‘/’), Potenz(‘^’). Zudem werden auch unäre arithmetische Funktionen wie die Wurzelfunktion `sqrt(x)` oder Betragsfunktion `abs(x)` unterstützt. Die Auswertung erfolgt hierbei rechts-assoziativ.
Die Reaktionsrate der ersten Regel aus Listing 3.3 oder die Initialisierungsanzahl aus Listing 3.7 könnten nun zum Beispiel wie in Listing 3.9 angegeben werden. Zudem wird eine wissenschaftliche

```
1 var  kInit = 100
2 init kInit (a: A, b: B): a.x//0, a.c(u)//0, b.y//0
3
4 var  sRate = sqrt(6*abs(-294.0))
5 rule simpleBinding(a: A, b: B): a.x//0, b.y//0 => a.x+b.y @sRate
```

Listing 3.9: Ausdrücke mit Variablen

Schreibweise für numerische Werte unterstützt. Dies ist insbesondere nützlich, da die Werte von Reaktionsraten oft sehr gering sind. Ein Beispiel für diese Schreibweise zeigt Listing 3.10. Neuzuweisungen von Werten sind **nicht** möglich. Alle definierten Variablen sind *final* und unterliegen *statischem Scoping*.

```
1 var normal = 0.00103
2 var scientific = 1.05E-3
```

Listing 3.10: Wert $1.05 \cdot 10^{-3}$ in *Re.action*

Observer

Alleine mit Modellsignatur, Initialisierungen und Regeln ist schon die Definition eines vollständigen Reaktionsmodells möglich – zumindest ohne konkrete Direktiven für die Simulation selbst. Um die Entwicklung der Konzentration bestimmter Moleküle und Molekülverbindungen im System über die Zeit zu ermitteln, benötigt man ein Feature, das definiert, welche Moleküle im System beobachtet und statistisch erfasst werden sollen.
Hierzu gibt es sogenannte *Observer*, die über das Schlüsselwort ‘observe’ eingeleitet werden und einen Namen sowie ein zu beobachtendes Muster besitzen. Ihre Form ist in Listing 3.11 dargestellt.

```
observe (<Signatur>): <Muster>
```

Listing 3.11: Struktur von Observern in *Re.action*

Ein konkretes Beispiel zu *Observern* zeigt Listing 3.12. Hier werden durch den Observer ‘aFree’ alle im Modell vorkommenden Agenteninstanzen vom Typ A mit ungebundenen Sites `x` und `c` beobachtet. Der Observer ‘a_cp’ hingegen findet alle Agenteninstanzen vom Typ A, deren Site `c` sich im Zustand `p` befindet. Der Bindungszustand dieser Site ist nicht spezifiziert, weswegen die Site sowohl gebunden,

```
observe aFree   (a: A): a.x//0, a.c//0
observe a_cp    (a: A): a.c(p)
```

Listing 3.12: Observers in *Re.action*

als auch frei sein kann. Dies ist insbesondere sinnvoll, da in der Regel nicht die Konzentrationen aller im Modell vorkommenden Agentenverbindungen interessieren, sondern nur bestimmter, ausgesuchter Agentenkonfigurationen bzw. Molekülarten.

Abbruchbedingungen

Um das Simulationsmodell zu vervollständigen, sind zuletzt noch Abbruchbedingungen nötig, damit die Simulation nicht unendlich lange ausgeführt wird.
Entsprechende Abbruchbedingungen werden über sogenannte `terminate`-Befehle definiert. Dazu lassen sich als Abbruchparameter sowohl die vergangene Simulationszeit in der für Reaktionsraten genutzten Einheit, wie z.B. Millisekunden, als auch die Anzahl ausgeführter Iterationen oder die Menge der aktuell vorhandenen Exemplare eines bestimmten Moleküls nutzen. Eine Iteration beschreibt hierbei das Finden aller Vorkommnisse von Mustern im Modell außer Nachbedingungen von unidirektionalen Regeln und Mustern aus Initialisierungen, sowie die Anwendung von Transformationsregeln.
Diese `terminate`-Befehle bestehen aus einer der in Listing 3.13 gezeigten Strukturen.

```
terminate time                                  = <Zeit>
terminate iterations                            = <Iterationsanzahl>
terminate (<Signatur>): <Muster> matches        = <Anzahl>
```

Listing 3.13: Struktur von Abbruchbedingungen in *Re.action*

Wie diese Befehle konkret aussehen können, zeigt Listing 3.14. Sollten die dort aufgeführten Befehle genau so in einem spezifizierten Modell auftauchen, wird die Simulation entweder nach einer Dauer von 10000 Zeiteinheiten, der Durchführung von 100 Iterationen oder dem Erreichen von 500 im Modell vorhandenen Agenten des Typs A terminieren, je nachdem welcher Fall zuerst eintritt.

```
terminate time =                10000   # nach 10000 Zeiteinheiten
terminate iterations =            100   # nach 100 Iterationen
terminate (a: A): a matches =     500   # bei 500 Agenten von Typ A
```

Listing 3.14: Abbruchbedingungen in *Re.action*

Komplexe und Kompaktifizierung

An den beschriebenen Sprachkomponenten der vorherigen Abschnitte wird deutlich, dass die komponenteneigenen Signaturen den Umfang der entsprechenden Konstrukte enorm vergrößert. Dies wird beispielsweise in Listing 3.3 deutlich. Die Regeln wären viel kompakter und lesbarer, wenn man die zugehörige Signatur nicht bei jeder Regel neu angeben müsste. Insbesondere wenn es zu Mustern mit vielen verschiedenen aktiven Instanzen oder zu Agententypen mit langen Typnamen bekommt, ist die Signatur eine Komponente, die besonders viel Platz einnimmt. Um dieses Problem zu umgehen und dem Grundsatz der Kompaktheit treu zu bleiben, wurden sogenannte *Komplexe* eingeführt. Diese Komplexe erhalten eine Signatur, die sich transitiv auf die darin enthaltenen Sprachkomponenten wie Regeln oder sogar weitere Komplexe überträgt. Das bekannte Regelset aus Listing 3.3 reduziert sich damit zu dem Ausdruck in Listing 3.15. Durch die einmalige Definition der Signatur zu Beginn des

```
1 complex (a: A, b: B){
2     rule simpleBinding:     a.x//0, b.y//0  => a.x+b.y      @1.0
3     rule simpleSplit:       a.x+b.y         => a.x//b.y     @1.0
4     rule stateChange:       a.c(u)          => a.c(p)       @1.0
5     rule synthesis:         _               => a            @1.0
6     rule degradation:       a               => _            @1.0
7 }
```

Listing 3.15: Reduziertes Regelset in Komplex

Komplexes, sind die Instanzen `a` und `b` in allen Regeln innerhalb des Komplexes abrufbar. Analog gilt dies in Komplexen auch für andere Komponenten, die eine Signatur benötigen, wie z.B. *Observer*. Somit kann die Signatur dieser Komponenten einfach ausgespart werden. Das Scope von in Komplexen definierten Variablen erstreckt sich jedoch nicht über den entsprechenden Komplex hinaus. Sie kann nur innerhalb desselben Komplexes referenziert werden, in dem sie auch definiert wurde.
Dies bedeutet allerdings nicht, dass es nun nicht mehr möglich wäre, weitere Instanzen zu den Regeln hinzuzufügen. Wenn für eine Regel, eine Initialisierung oder einen Observer eine weitere Instanz benötigt wird, kann man diese ganz normal wie in der üblichen Signatur definieren. Die komponenteneigene Signatur wird dann zur Komplexsignatur hinzugefügt. Dies zeigt beispielsweise Listing 3.16.
Ebenso ist es möglich, Komplexe beliebig tief zu schachteln. Auch hier fügt sich die Signatur in jeder

```
1 complex (a: A){
2     rule simpleBinding (b: B):  a.x//0, b.y//0  => a.x+b.y      @1.0
3     rule simpleSplit (b: B):    a.x+b.y         => a.x//b.y     @1.0
4     rule stateChange:           a.c(u)          => a.c(p)       @1.0
5     rule synthesis:             _               => a            @1.0
6     rule degradation:           a               => _            @1.0
7 }
```

Listing 3.16: Reduziertes Regelset in Komplex mit komplexinternen Signaturen

Ebene aus allen Signaturen der darüber liegenden Komplexe zusammmen. Das Regelset aus Listing 3.16 könnte also ebenfalls wie in Listing 3.17 gezeigt definiert werden. Ein weiteres Feature zur Kompaktifizierung ist das Übertragen von Bindungsverboten über den '`//`'-Operator auf alle Sites eines Agenten. So kann sich beispielsweise das Muster des Observers `aFree` aus Listing 3.12 zum Ausdruck in Listing 3.18 verkürzen.

```
1 complex (a: A){
2     complex (b: B){
3         rule simpleBinding:     a.x//0, b.y//0  => a.x+b.y      @1.0
4         rule simpleSplit:       a.x+b.y         => a.x//b.y     @1.0
5     }
6     rule stateChange:           a.c(u)          => a.c(p)       @1.0
7     rule synthesis:             _               => a            @1.0
8     rule degradation:           a               => _            @1.0
9 }
```

Listing 3.17: Reduziertes Regelset mit geschachtelten Komplexen

```
    observe aFree   (a: A): a//0
```

Listing 3.18: Kurzschreibweise für freie Agenten

3.4. Modellarchitekturen

Das aus der Grammatik der zuvor gezeigten Sprache hervorgehende Modell ist nur eins von drei verschiedenen Modellen, die zur Einbindung des Frameworks in das Simulationstool *SimSG* notwendig sind. Dazu zählen außerdem ein Arbeits- bzw. Zwischenmodell, auf dem die essentiellen Informationen des DSL-Modells für die abschließende Transformation aufbereitet werden, und zuletzt das ausführbare Simulationsmodell, wie bereits in Abschnitt 3.1 beschrieben. Dieser Abschnitt widmet sich dem Aufbau dieser Modelle und den Transformationen zwischen ihnen. Der allgemeine Work Flow wird hierbei durch Abbildung 3.7 beschrieben.

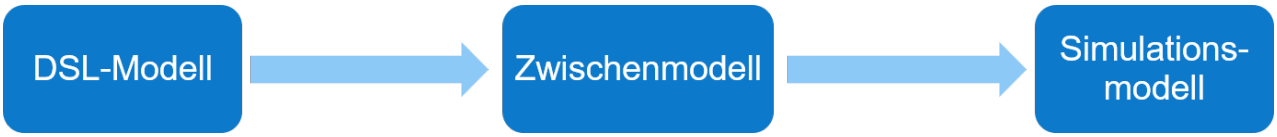
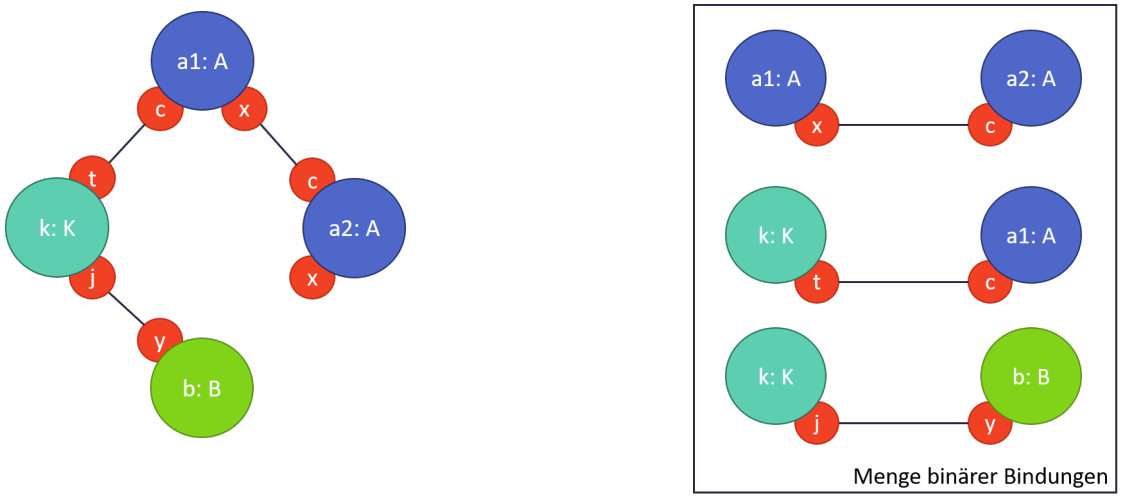


Abbildung 3.7.: Pipeline der Modelltransformationen

3.4.1. Sprachmodell

Das Sprachmodell wird automatisch von *Xtext* generiert und direkt aus der Grammatik der Sprache abgeleitet, die in Appendix A aufgeführt ist. Dabei werden die verschiedenen Komponenten einer konkreten Spezifikation durch Objekte entsprechender (Java-)Klassen beschrieben. Eine Transformationsregel zum Beispiel wird durch ein Regel-Objekt repräsentiert, das wiederum eigene Komponenten, wie unter anderem die Regelsignatur oder die Vor- und Nachbedingungen beinhaltet.
Das Metamodell der Sprache, das alle möglichen Ausdrücke und Formulierungen abdeckt, besteht aus Klassen, die die entsprechenden Objekte beschreiben. Das Verhältnis der Klassen untereinander wird durch Referenzen zu anderen Klassen und unter Ausnutzung von Polymorphie in der Vererbungshierarchie abgebildet. So besteht beispielsweise das gesamte Modell aus einer Menge bestimmter Komponenten, die durch die Sprache *Re.action* definiert werden können. Diese Komponenten werden

über Klassen für Agentendefinitionen, Initialisierungen, Variablen, Regeln, Observer und Terminate-Befehle repräsentiert. All diese definierbaren Komponenten erben von einer Superklasse *Component*, sodass für jede Komponente jede der genannten Subklassen eingesetzt werden kann.
An allen anderen Stellen, wo in der Grammatik von *Re.action* verschiedene Komponentenarten erwartet werden können, wie zum Beispiel eine konkrete Agenteninstanz oder nur ein generischer Agententyp auf der rechten Seite einer Bindung, wird dasselbe Prinzip angewandt.
Alle in der Sprache definierten Muster werden hier noch genau so repräsentiert, wie sie direkt in der Spezifikation dargestellt werden. Das in Abbildung 3.8a gezeigte Muster wird somit beispielsweise durch eine Summe mehrerer Bindungen der beteiligten Sites repräsentiert, wie es Abbildung 3.8b verdeutlicht.



(a) Beispielmuster mit neuem Agententyp K

(b) Darstellung des Musters im Sprachmodell

Abbildung 3.8.: Repräsentation von Mustern im Sprachmodell

3.4.2. Zwischenmodell

Um dieses Sprachmodell in das Simulationsmodell zu überführen, könnte man in einer naiven Implementierung eine einfache, direkte Transformation aus dem Sprachmodell zum Simulationsmodell nutzen und das Zwischenmodell aus Abbildung 3.7 auslassen. Dies ist jedoch aus vielerlei Gründen nicht ratsam, weswegen hier ein Arbeitsmodell als Zwischenstufe der Gesamttransformation etabliert wird.
Das Fundament der Transformation mit Zwischenmodell bildet der bereits 1982 von Edsger Dijkstra definierte *separation of concerns* [32], nach dem verschiedene Aspekte eines Systems immer isoliert betrachtet werden sollten. Dies erhöht die Wartbarkeit des gesamten Frameworks enorm, da Änderungen separat an den zwei Transformationsschritten vorgenommen werden können. Im besten Falle führt eine Änderung der Sprache sogar dazu, dass nur die Transformation zum Arbeitsmodell angepasst werden muss und der zweite Transformationsschritt vom Arbeitsmodell zum Simulationsmodell nach keinerlei Anpassung mehr verlangt, da das Arbeitsmodell ja unverändert bleibt. Somit wird auch die Erweiterbarkeit der Sprache durch einen zweigeteilten Transformationsprozess deutlich erhöht.
Des Weiteren wird eine gewisse Modularität erhalten, sodass man an die Spezifikation leichter weitere

Simulations- oder Pattern-Matching-Engines anschließen kann, da nur der zweite Transformationsschritt in Hinblick auf die neue Engine angepasst werden muss.
Ziel des Arbeitsmodells ist es außerdem, Redundanzen, die durch die Grammatik und Syntax der Sprache im Sprachmodell entstehen, zu entfernen und der Transformation zum Simulationsmodell schon etwas entgegen zu arbeiten, indem diverse Repräsentationen angepasst werden. Da in der Simulation jede Agenteninstanz als eigene Entität betrachtet wird, deren Sites entweder gebunden oder ungebunden sind, ist es sinnvoll, eine Darstellung für Agenteninstanzen abzuleiten, die nicht die in der Spezifikation definierten binären Bindungen zwischen jeweils zwei Sites in den Mittelpunkt stellt, sondern die konkrete Agenteninstanz als solche. Hierzu wird die Menge an Bindungen im Arbeitsmodell in eine Menge aus Agenteninstanzen überführt, deren Sites sich gegenseitig als Bindungspartner referenzieren. Das Metamodell zur Darstellung von Patterns zeigt Abbildung 3.9 und repräsentiert Muster nun nicht mehr über eine Menge aus zweiwertigen Bindungen zwischen Sites, sondern über eine Menge aus am Muster beteiligten Agenteninstanzen, die Referenzen auf Instanzen ihrer Sites enthalten, welche wiederum Referenzen auf ihre Zustände und andere Site-Instanzen besitzen, mit denen sie verbunden sind oder konkret nicht verbunden sein sollen. Alle Pattern sind in einem *PatternContainer* enthalten, der alle Muster verwaltet. Die in Regeln, Initialisierungen, Observern und Terminate-Befehlen nach Musteranzahl enthaltenen Muster sind nur Verweise auf die entsprechenden Muster im *PatternContainer*.
Das vollständige Klassendiagramm zum Zwischenmodell befindet sich in Appendix B in Abbildung **??**.

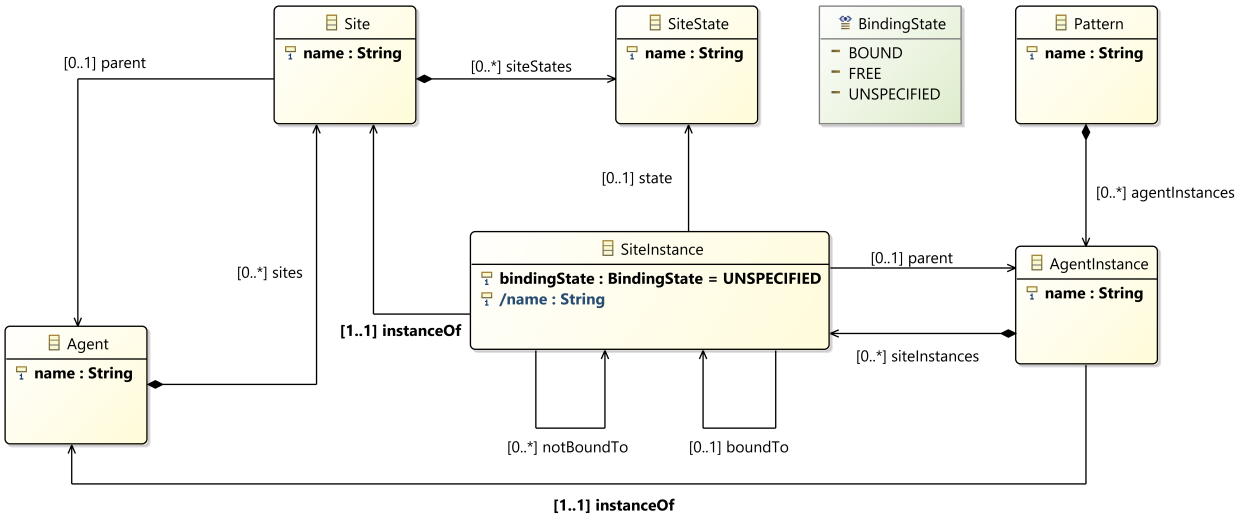


Abbildung 3.9.: Metamodell von Mustern im Arbeitsmodell (Auszug)

Agent und *Site* repräsentieren die im Modell definierten Agententypen mit den von ihnen besessenen Sites und deren möglichen *SiteStates*. Zusammen bilden diese Klassen also alle in Mustern nutzbaren Konfigurationen der definierten Agententypen. Die Klassen *AgentInstance* und *SiteInstance* bilden nun konkrete Instanzen dieser Menge an möglichen Konfigurationen ab, indem sie deren Bindungszustände und interne Zustände in Mustern der Spezifikation, wie sie beispielsweise in Vor- und Nachbedingungen von Regeln auftauchen, konkretisieren. Sie beschreiben also konkrete Molekülentitäten und nicht nur den Typen eines Agenten. Zudem besitzen sie immer eine Referenz zu ihren jeweiligen Elternagenten und -Sites (*instanceOf*-Referenzen). Hierbei handelt es sich noch **nicht** um eine Repräsentation der Agentenknoten des späteren Simulationsmodells. Alle Agenteninstanzen einer Komponente sind hierbei als *AgentInstance* in einem zugehörigen Pattern enthalten. Jede Agenteninstanz ist abgeleitet von

einem Agententypen *Agent* und enthält wiederum konkrete Instanzen seiner existierenden Sites. Diese Site-Instanzen besitzen sowohl ein Attribut *bindingState*, welches die über das Enum *BindingState* definierten, selbsterklärenden Bindungszustände `FREE`, `BOUND` oder `UNSPECIFIED` annehmen kann, als auch einen Namen, sowie Referenzen zu ihrem internen Zustand, einer Menge an *SiteInstances*, zu denen die Bindung verboten wurde und einer verbundenen *SiteInstance*. All diese Komponenten müssen nicht existieren, falls eine Site zum Beispiel keine internen Zustände besitzt oder frei ist. In diesem Fall verweisen die entsprechenden Referenzen auf `null` bzw. eine leere *Collection*.
Außerdem werden essenzielle Informationen aus den Redundanzen gefiltert, die sich aus der Grammatik im Sprachmodell ergeben. Beispielsweise werden alle spezifizierten Muster extrahiert, in eine Form entsprechend Abbildung 3.9 überführt und ihrer übergeordneten Komponente (Regel, Initialisierung, Observer oder Terminate-Befehl nach Musteranzahl) zugeordnet. Informationen, wie die Schachtelung von den in Abschnitt 3.3.2 eingeführten Komplexen, in denen sich die Regel befand, werden aussortiert. Ebenso werden bidirektionale Regeln direkt in zwei unidirektionale Regel für Hin- und Rückrichtung der Transformation aufgelöst, sowie alle arithmetischen Ausdrücke direkt evaluiert und als ausgewertete Attribute im *Integer*- oder *Double*-Datenformat in den jeweiligen Objekten der Klassen abgespeichert.

Export in BNGL-Format Die Aufspaltung der Gesamttransformation in zwei Subtransformationen bietet zudem den Vorteil, bei dem Hinzufügen von neuen Features wählen zu können, ob diese aufgrund der Strukturen des ursprünglichen Sprachmodells oder des Zwischenmodells implementiert werden. So wurde es durch das Zwischenmodell einfacher, die Transformation und den Export von in *Re.action* formulierten Modellen zu anderen etablierten Sprachformaten wie zum Beispiel der *BioNetGenLanguage* zu realisieren. Hierbei ist die Repräsentation des biochemischen Systems im Zwischenmodell viel näher an der in *BNGL* gewählten Repräsentation als das ursprüngliche Sprachmodell. Jedes in *Re.action* kann durch Auswahl der entsprechenden Option bei Bedarf in das *BNGL*-Format exportiert werden.
Dieses Feature bringt viele Vorteile mit sich, da dasselbe Modell nun auch in anderen Simulatoren genutzt werden kann, die das *BNGL*-Format unterstützen, und somit verschiedene Simulatoren effektiv miteinander verglichen werden können. Die Integration weiterer Exportmöglichkeiten in den Transformations-Workflow ist analog zu dem bereits implementierten Feature ein Leichtes.

3.4.3. Simulationsmodell

Das aus dem Zwischenmodell entstehende Simulationsmodell setzt sich nun aus einer *SimulationDefinition* und sogenannten *IBeX*- und *GT*-Modellen zusammen. Die *SimulationDefinition* enthält das aus konkreten Agenteninstanzen und möglichen internen Zuständen bestehende Modell, auf dem die Simulation arbeiten soll, sowie Referenzen auf die *IBeX*- und *GT*-Modelle. Diese sind für das Funktionieren der in *SimSG* genutzten Pattern Matcher zwingend notwendig.
Sie repräsentieren die im Sprachmodell definierten Regeln und Muster für die tatsächliche Simulation und werden ebenfalls in der *SimulationDefinition* referenziert, wo allen Regeln auch Reaktionsraten zugewiesen werden und sich die Abbruchbedingungen der Simulation befinden. Für die Erstellung der GT- und IBeX-Modelle und zur Generierung eines konkreten Modells aus den spezifizierten Initialisierungen, auf dem die Simulation operieren soll, wird nun ein Modell benötigt, das alle spezifizierten Muster auch sinnvoll als Graphen abbilden kann. Hierzu wird eine dreistufige Modellhierarchie definiert, die in Abbildung 3.10 gegeben ist und sich erneut exemplarisch an dem bisher genutzten Beispiel der zwei Agententypen A und B orientiert.
An der Spitze der Hierarchie steht ein statisches Metametamodell, das den allgemeinen Aufbau eines konkreten Simulationsmodells definiert. Jedes Simulationsmodell besteht aus einem Container,

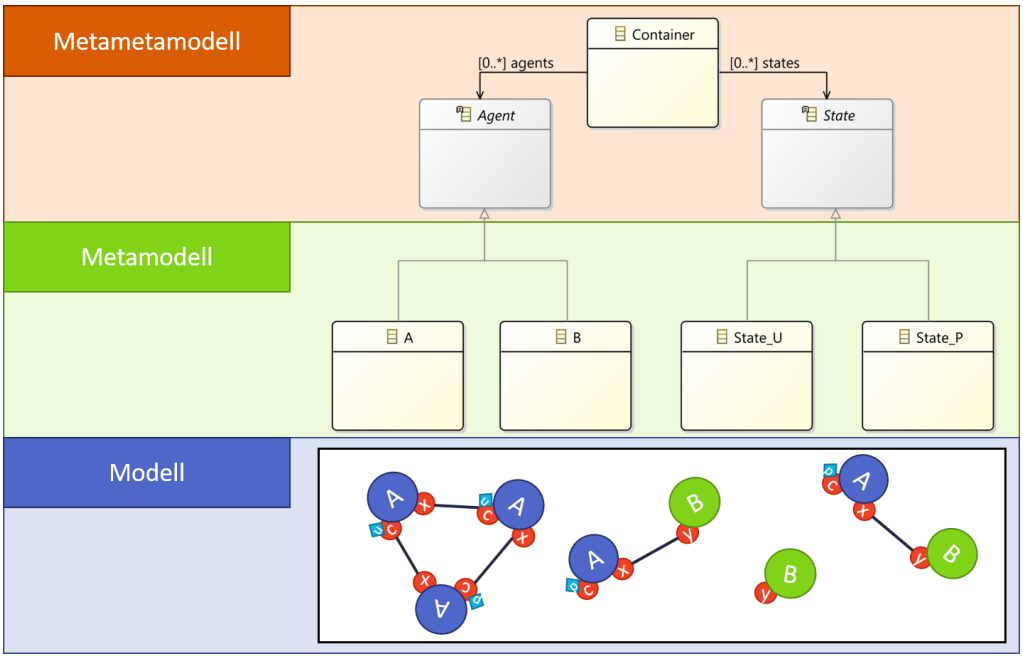


Abbildung 3.10.: Modellhierarchie des Simulationsmodells

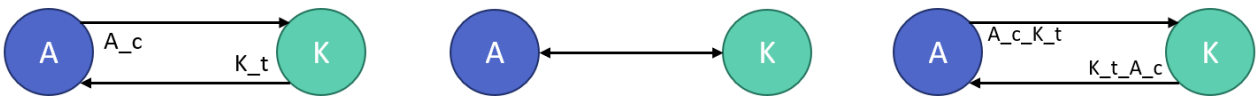
der eine endliche Menge an Agenten und Zuständen beinhaltet. Hierbei gilt es zu beachten, dass in den Simulationsmodellen keine Unterscheidung zwischen Agenten und Agenteninstanzen mehr vorgenommen wird. Spricht man hier von einem Agenten, beschreibt dies eine Molekülentität, wie es die Agenten- und Site-Instanzen im Zwischenmodell tun, und nicht den Typen einer Agenteninstanz. Alle Instanzen des hier gezeigten Modells werden im Graphen der Simulation später als Objekt der entsprechenden Klasse einen eigenen Knoten repräsentieren, weswegen im Folgenden die Begriffe Objekt, Instanz und Knoten austauschbar sind.

Zunächst wird aus den Regeln des Arbeitsmodells die Menge an tatsächlich genutzten Agenten- und Zustandstypen abgeleitet, für die jeweils eine eigene Klasse im Metamodell erstellt wird. Die Klassen zur Repräsentation von Agenten erben hierbei von der *Agent*-Klasse des Metametamodells und die Klassen zur Darstellung der Zustände von der entsprechenden *State*-Klasse.
Die Agentenklassen erhalten zusätzlich Referenzen für jeden möglichen Zustand, in dem sie sich befinden können. Befindet sich ein Agent in Zustand `u`, so wird eine entsprechende Referenz erstellt, die auf ein Objekt der Klasse `State_U` zeigt, die wiederum von der *State*-Klasse geerbt hat. Hierbei sind die Referenzen stark typisiert. Eine Zustandsreferenz kann nicht auf jedes beliebige von *State* erbende Objekt zeigen, sondern nur auf Objektinstanzen der Zustandsklasse, die die Referenz beschreibt. Betrachtet man die Site `c` eines Agenten des Typs A, die sich in Zustand `u` oder Zustand `p` befinden kann, so werden dazu im Metamodell zwei von *State* erbende Klassen erstellt, welche die beiden Zustände beschreiben. Namentlich konform zu Abbildung 3.10 sind dies *State_U* und *State_P*. In der Klasse des Agenten A werden nun zwei Referenzen erstellt, welche zusammen den aktuellen internen Zustand der Site `c` beschreiben. Konkret sind dies eine Referenz „`A_c_u`“, welche nur auf Objekte der Klasse *U_s* verweisen, und eine Referenz „`A_c_p`“, die ausschließlich Objekte der Klasse *P_s* referenzieren kann. Dies geschieht für alle Zustände, in denen sich die Sites eines Agenten befinden können. Dabei darf von den Referenzen, die die verschiedenen internen Zustände einer Site eines Agenten repräsentieren, immer nur eine aktiv sein, da sich eine Site immer nur in einem einzigen,

eindeutigen internen Zustand befinden kann. Weitere Referenzen, die auf andere States zeigen, dürfen nicht existieren. Dass dies stets der Fall ist, wird durch die Korrektheit der Modelltransformationen in Verbindung mit der Tatsache, dass in der *Re.action*-Sprache einer Site-Instanz maximal ein interner Zustand zugewiesen werden kann, gewährleistet.
Generell werden Sites nicht mehr durch eigene Objekte einer Site-Klasse repräsentiert, wie es noch im Zwischenmodell der Fall war, sondern implizit durch Referenzen der zugehörigen Agentenklasse definiert. Zur Darstellung von Sites und deren Verbindungen untereinander wurde zuerst eine naive Implementierung erstellt, bei der eine Bindung durch zwei unidirektionale Referenzen von Agenten auf sich gegenseitig realisiert wurden. Die Bindung `a.c+k.t` wäre also durch eine Referenz „`A_c`“ in der Klasse A auf die Klasse K und eine zweite Referenz „`K_t`“ in der Klasse K auf die Klasse A realisiert worden.
Im Laufe der Zeit stellte sich jedoch heraus, dass dieses Modell weder eindeutig noch vollständig ist, da hierbei die Information verloren geht, welche Sites miteinander verbunden sind.
Dies veranschaulicht folgendes Beispiel, das erneut den aus vorherigen Beispielen bekannten Agententypen A und den in Abbildung 3.8 neu eingeführten Typen K mit den Sites `t` und `j` enthält. In dem zuvor beschriebenen Modell würden für die Verbindungen der Sites `t` mit `x` und `j` mit `c` in beiden Agenten die entsprechenden Referenzen „`A_x`“, „`A_c`“, „`K_t`“ und „`K_j`“ existieren, die alle auf die Klasse des jeweils anderen Agenten verweisen. Hierbei wären jedoch beide in Abbildung 3.11 gezeigten Geometrien möglich, die eine durchaus wichtige Eigenschaft von Molekülkomplexen darstellen können, und somit unbedingt eindeutig sein müssen. Da sich ein Molekül nicht in zwei Geometrien gleichzeitig

Abbildung 3.11.: Beide möglichen Geometrien aus nicht eindeutigem unidirektionalem Modell

befinden kann, wurde ein neues Modell entwickelt, das die Verbindungen auf eindeutige Art und Weise speichert. Es ist damit zwar komplexer, verspricht aber potenzielle Performance-Verbesserungen auf Pattern Matching-Ebene.
In diesem neuen Modell werden die Referenzen auf gebundene Sites nun eindeutig durch den Namen der Referenz festgelegt. So wird eine Verbindung der Site `c` eines Agenten von Typ A mit einer Site `t` eines Agenten des Typs K nun nicht mehr wie in Abbildung 3.12a durch zwei Referenzen „`A_c`“ und „`K_t`“ erstellt, die gegenseitig auf die Agenten verweisen, sondern nur über eine einzelne bidirektionale Referenz zwischen den Agenten A und K wie in Abbildung 3.12b. Diese ergibt sich EMF-intern wieder über zwei einzelne, unidirektionale Referenzen, auf die in K über „`K_t_A_c`“ und in A über „`A_c_K_t`“ zugegriffen werden kann, wie Abbildung 3.12c verdeutlicht. Somit können die an der



(a) Unidirektionale Kanten (b) Bidirektionale Kante (c) Realisierung der bidirektionalen Kante

Abbildung 3.12.: Verschiedene Bindungsrealisierungen mit uni- und bidirektionalen Kanten

Verbindung beteiligten Sites und Agenten eindeutig definiert werden und die Referenzen werden nun bidirektional interpretiert. Zudem sind die Referenzen auf Sites nun wie bereits die Zustandsreferenzen stark typisiert. Im unidirektionalen Modell verwies jede Referenz auf die Klasse *Agent* des Metametamodells aus Abbildung 3.10, sodass sie auf jeden beliebigen Agenten verweisen konnte. Im neuen, bidirektionalen Modell gibt es für jede mögliche Verbindung zwischen Sites eine eigene Referenz. Da somit für jede Referenz eindeutig festgelegt ist, auf welche Site und somit auch auf welche Agentenklasse sie verweist, kann sie auf den entsprechenden Agententypen eingeschränkt werden.
Durch diese Neuerung vergrößert sich der Umfang des Metamodells potenziell enorm, wenn für jede Site eines Agenten eine Referenz zu jeder anderen Site erstellt wird. Dies ist äquivalent zu der Anzahl Kanten in einem vollständigen Graphen mit n Knoten, indem alle Knoten über eine Kante miteinander verbunden sind, sodass jeder Knoten alle anderen $n-1$ Knoten als Nachbarn besitzt, wobei jede dieser Kanten eine eigene Referenz im Metamodell darstellt. Die Anzahl der Referenzen dieses neuen Metamodells ergibt sich ausgehend von n Sites im Modell insgesamt aus der Anzahl aller möglichen ungeordneten Paare aus Sites über den Binomialkoeffizienten wie in Gleichung (3.1).

$$\binom{n}{2} = \frac{n(n-1)}{2} \tag{3.1}$$

Für die verschiedenen Modelle ergibt sich damit die in Tabelle 3.1 gezeigte Komplexität in Bezug auf die Zahl der zu erstellenden Referenzen. Tatsächlich wurden bisher im unidirektionalen Modell stets alle n möglichen Referenzen erstellt. Im bidirektionalen Modell werden nicht zwingend alle theoretisch möglichen Referenzen erstellt, sondern – um dem entgegenzuwirken – alle tatsächlich auftretenden Bindungen zwischen Sites und die dafür nötigen Referenzen aus der Menge aller Regeln abgeleitet und nur diese auch im Metamodell generiert.

	Unidirektionales Modell	Bidirektionales Modell
Zu erstellende Referenzen	n	$\frac{n(n-1)}{2}$
Komplexitätsklasse	$O(n)$	$O(n^2)$

Tabelle 3.1.: Komplexität der Referenzen- bzw. Kantenzahl im Metamodell

Durch die weitere Übersetzung des Metamodells mit all seinen Klassen und Referenzen in ein entsprechendes *GT*- und *IBeX*-Modell ergibt sich ein konkreter gerichteter Graph, auf dem die Pattern Matcher während der Simulation operieren.
Hierzu wird für jede existierende Agenteninstanz des Modells ein Objekt seiner entsprechenden Agentenklasse erzeugt und als Knoten interpretiert. Die Kanten des Graphen werden durch die Referenzen der Agentenobjekte auf andere Objekte definiert. Dabei wird für jeden Agenten im Modell ein eigenes Objekt als Knoten erzeugt, für jeden möglichen internen Zustand von Sites allerdings nur einzelnes Zustandsobjekt bzw. auf Graphenebene nur ein einzelner Zustandsknoten. Das für einen konkreten Site-Zustand referenzierte Zustandsobjekt, um den internen Zustand der Sites von Agenten anzugeben, ist also für gleiche Zustände stets dasselbe. Alle Agentenknoten mit Sites im internen Zustand ‘p’ werden mit dem einen Zustandsknoten verbunden sein, der den entsprechenden internen Zustand ‘p’ repräsentiert. Dies hilft ebenfalls, den Umfang des Modells durch weniger Knoten gering zu halten.
Um dem Pattern Matcher die verschiedenen im Sprachmodell definierten Muster und Transformationsregeln nun zur Verfügung zu stellen, müssen sie in *GT*-Regeln und *IBeX*-Patterns übersetzt werden, die die Muster modellhaft in *eMoflon* repräsentieren.

GT-Modell

Das GT-Modell besteht hierbei aus einer Menge an GT-Regeln, die jeweils über sogenannte GT-Knoten definieren, welche Knoten überhaupt an einer Regel beteiligt sind. Hierzu werden alle Knoten eines Musters bijektiv auf die GT-Knoten der neu erstellten GT-Regeln abgebildet. Jede der GT-Regeln enthält nun einen GT-Knoten für jeden relevanten Knoten des Musters, also die entsprechenden Agenten- und Zustandsknoten. Der Typ eines Knotens wird definiert, indem ihm eine entsprechende Klasse aus dem Metamodell zugeordnet wird. So kann über die *eMoflon*-API Java-Code aus dem GT-Modell generiert werden.

IBeX-Modell

Das IBeX-Modell definiert die genauen Transformationen und Beziehungen der Knoten untereinander. Hierzu existieren *ContextPatterns*, die ein Muster definieren, das später gefunden werden muss; also zum Beispiel die Vorbedingung einer Regel oder das beobachtete Muster eines Observers. Für Reaktionsregeln kommen sogenannte *DeletePatterns* und *CreatePatterns* hinzu, die definieren, welche Komponenten aus dem gefundenen Match entfernt oder diesem hinzugefügt werden.
Zur Repräsentation der Knoten eines Musters werden sogenannte *IBeXNodes* definiert, die analog zu den GT-Knoten einen Typen besitzen, der auf eine entsprechende Klasse aus dem Metamodell verweist. Für die Kanten werden sogenannte *IBeXEdges* generiert, welche zwei *IBeXNodes* enthalten, die miteinander verbunden sind. Diese *IBeXEdges* besitzen ebenfalls einen Typen. Dieser besteht aus einer Referenz im Metamodell entsprechend den Typen bzw. Metamodellklassen der beiden verbundenen Knoten. Will man nun freie Sites in einem solchen *ContextPattern* definieren, geschieht dies mittels der in Abschnitt 2.3.3 eingeführten *Negative Application Conditions (NACs)*. Da eine freie Site letztlich nichts anderes bedeutet, als dass der Knoten des Agenten, zu dem sie gehört, keine Kanten zu Agentenknoten besitzt, werden alle möglichen Kanten, welche die entsprechende Site repräsentieren, zu anderen Knoten via NACs verboten. Hierzu werden zunächst alle möglichen Referenzen im Metamodell gesucht, die eine Kante von der betroffenen Site zu einer anderen Site darstellen. Für jede dieser Referenzen wird ein beispielhaftes Muster erstellt, dass nur aus einer Verbindung der beiden Sites besteht. All diese erstellten, „verbindenden“ Muster werden nun jeweils als NAC im *ContextPattern* mit dem als frei definierten Knoten aufgerufen, sodass dieser mit **jedem** anderen potenziellen Bindungspartner **nicht** verbunden ist. Somit werden alle möglichen Bindungen, welche die Site potenziell eingehen könnte, verboten, sodass nur Matches gefunden werden können, in denen sie tatsächlich ungebunden und somit *frei* ist.

Unterspezifizierte Bindungen Will man hingegen keine freie Site definieren, sondern eine unterspezifizierte Site, die mit einem beliebigen, unbekannten Agenten wie in Abbildung 3.2c verbunden ist, müsste man für jeden möglichen Bindungspartner der Site eine *Invocation* erstellen und diese Invocations mit einem logischen *ODER* verknüpfen. Dies ist jedoch mit klassischem Pattern Matching nicht möglich, weswegen hierzu die *DeMorgansche Regel* angewandt wird. Dabei wird eine Schachtelung von NACs wie in Abbildung 3.13 auf logischer Ebene als Schachtelung von Negationen betrachtet. Geht es beispielsweise darum, dass die Site x eines Agenten A in einem Pattern P mit einem beliebigen Agenten verbunden sein soll, so wird zunächst ein Muster erstellt, das mittels NACs alle möglichen Verbindungen der betroffenen Site verbietet. Dieses Muster definiert die Site also als *freie Site*. In

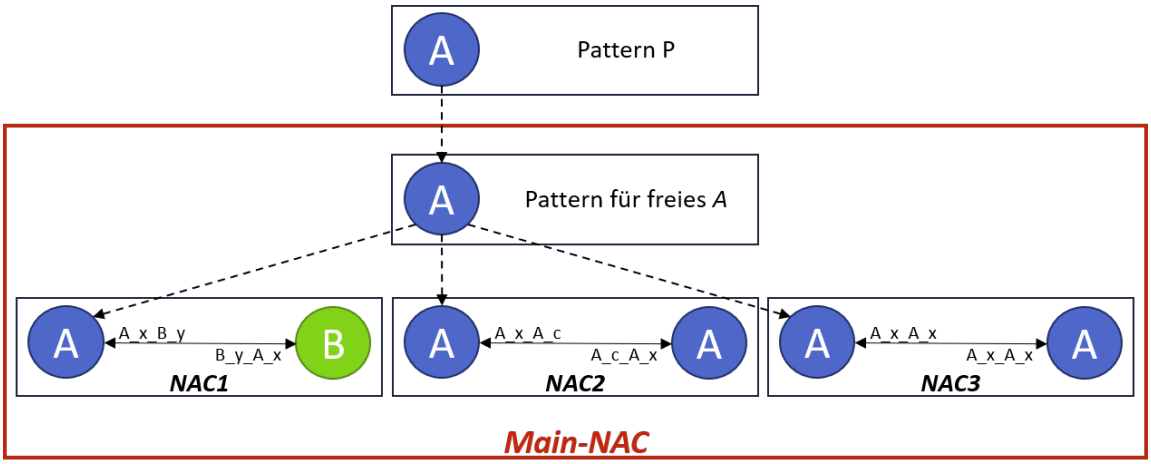
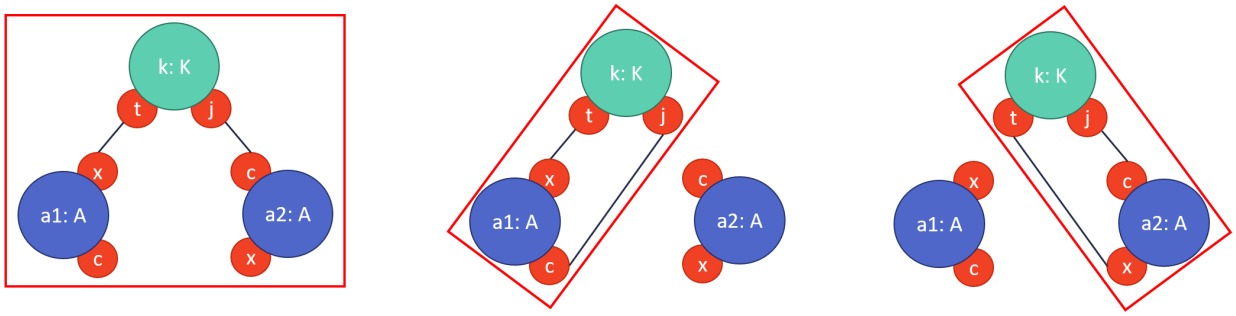


Abbildung 3.13.: Pattern P mit unterspezifizierter Site x eines Agenten A als *IBeX*-Pattern

diesem Beispiel werden dazu die NACs 1 bis 3 erstellt, welche die Verbindungen zwischen den Sites x, c und y verbieten – unter der Annahme, dass damit alle Sites von im System vorhandenen Agenten abgedeckt sind. Im Anschluss wird eben dieses Muster wiederum als „Haupt-NAC“aufgerufen. Durch diese Schachtelung aus NACs ergibt sich, dass die Site über eine der potenziell möglichen Kanten eine Verbindung zu einem weiteren Agenten besitzen **muss**.

Injektivitätsbedingungen Des Weiteren müssen *ContextPatterns*, die mehrere Knoten desselben Typs enthalten, Injektivitätsbedingungen zugewiesen werden, um zu definieren, dass es sich bei den im Modell enthaltenen Knoten tatsächlich um verschiedene Knoten handeln soll. Warum diese Einschränkung notwendig ist, zeigt Abbildung 3.14. Hier wird eigentlich das Muster aus Abbildung 3.14a gesucht. Dieses Muster ist bei korrekter Definition der Injektivitätsbedingungen in seiner Form einzigartig. Ohne Injektivitätsbedingungen kann jedoch derselbe Agentenknoten mehrmals für Knoten desselben Typs im Muster verwendet werden, wie es in den Abbildungen 3.14b bzw. 3.14c der Fall ist. Dort würden ohne Definition von Injektivitätsbedingungen im Muster auch Matches gefunden werden, in denen der Knoten des Agents a1 bzw. a2 gar nicht vorhanden ist, da der jeweils andere Knoten doppelt genutzt wird.



(a) (b) (c)

Abbildung 3.14.: In (b) und (c) sind weitere mögliche Matches zu dem Muster aus (a) ohne Injektivitätsbedingungen an a1 und a2 rot hervorgehoben

Leere Muster Unter anderem aus den Vorbedingungen von Syntheseregeln können leere Muster hervorgehen. Diese werden im Zwischenmodell noch als Pattern mit fehlenden Agenteninstanzen repräsentiert. Analog dazu würde man in den *IBeX*-Patterns nun ein Muster erwarten, das keinerlei *IBeXNodes* enthält. Da *SimSG* dies jedoch nicht unterstützt, wird dem Muster der Container-Knoten hinzugefügt, der alle im System enthaltenen Agenten- und Zustandknoten enthält, wie schon Abbildung 3.10 zeigte. Dies garantiert in jeder Iteration der Simulation ein einzelnes Match, auf das direkt die in der Syntheseregel definierte Reaktionsrate angewandt werden kann, da der Container immer der Wurzelknoten des Systems und somit mindestens und maximal einmal vorhanden ist.

Vorteile des bidirektionalen Modells In Bezug zum Pattern Matching offenbart sich nun auch der Vorteil des bidirektionalen Modells mit starker Typisierung. Durch die Umstellung unidirektionaler Kanten auf bidirektionale Kanten muss der Pattern Matcher in jedem Muster weniger solche Kanten überprüfen. Im unidirektionalen Modell konnte jede Referenz bzw. Site mit allen anderen Knoten des Modells verbunden sein, da jede Referenz unter Ausnutzung von Polymorphie stets auf Knoten des Typs *Agent* aus dem Metametamodell in Abbildung 3.10 zeigte, von dem alle konkreten Agentenknoten erbten. Daher mussten zur Prüfung der Verbundenheit zweier Sites immer alle von einem Knoten ausgehenden Kanten überprüft werden, ob sie Kandidaten für ein potenzielles Match sind. Darunter fällt unter anderem, ob deren Zielknoten auch eine entsprechende Kante zurück auf den eigentlichen Ursprungsknoten besitzen, da eine Verbindung zwischen zwei Sites im IBeX-Pattern hier durch zwei Kanten repräsentiert wurde. Im neuen, bidirektionalen Modell sind alle Referenzen eines Agenten eindeutig einer Site des Bindungspartners zugewiesen, wodurch sich die Zahl der zu überprüfenden Kanten eines Knoten zunächst auf die Menge der Kanten reduziert, die den entsprechenden Typ aufweisen. Dies ist die erwähnte starke Typisierung. Des Weiteren muss nach dem Finden einer solchen Kante nicht zusätzlich überprüft werden, ob deren Zielknoten ebenfalls eine Kante in Rückrichtung besitzen, die auf den ursprünglichen Knoten als Zielknoten verweist. Aufgrund der bidirektionalen Kanten reicht nur eine einzige Kante in den IBeX-Patterns aus, um eine Verbindung zwischen zwei Sites zu repräsentieren.

4. Evaluation

Da das Modellierungs-Framework nun vollständig implementiert und erklärt ist, muss dessen allgemeine Effektivität und praktischer Nutzen bewertet werden.
Der erste Teil des folgenden Kapitels eruiert, ob die in Abschnitt 3.3.1 dargelegten Ziele der Spezifikation erreicht werden konnten. Hierzu werden in Abschnitt 4.1 zunächst die Stärken und Schwächen von *Re.action* diskutiert und daraufhin konkret mit anderen Sprachen der Biochemiedomäne verglichen. Abschließend werden die Ergebnisse knapp zusammengefasst und beurteilt, unter welchen Umständen die verschiedenen evaluierten Sprachen eine gute bzw. schlechte Wahl zur Modellierung darstellen.
Da ein Großteil des Frameworks beziehungsweise die Ausführung der Simulation selbst auf *Pattern Matching* basiert, ist es naheliegend, die Performance verschiedener Pattern Matcher gegeneinander abzuwägen. Hierzu werden konkret im zweiten Abschnitt 4.2 die beiden in *eMoflon* für Pattern Matching genutzten Tools *Democles* und *HiPE* in Bezug auf ihre Laufzeit und ihren Speicherverbrauch in verschiedenen Modellen evaluiert. Dabei werden auch stets die beiden in Abschnitt 3.4.3 vorgestellten Metamodelle mit unidirektionalen bzw. bidirektionalen Kanten und ihre Auswirkungen auf die genannten Parameter abgewägt.

4.1. Diskussion der DSL

4.1.1. Usability

Um die *Usability* einer Sache beurteilen zu können, muss zunächst der Begriff geklärt werden. *Usability* beschreibt die allgemeine „Nutzbarkeit “ des betrachteten Produkts. Die dazu beitragenden Parameter entspringen einem breiten Spektrum der generellen Benutzerfreundlichkeit beziehungsweise im Kontext dieser Arbeit der Software-Ergonomie des Frameworks. Der Begriff der Software-Ergonomie stellt insbesondere im hier wesentlichen Spezifikationskontext die intuitive Verständlichkeit der Sprache, sowie die kompakte Möglichkeit zur Eingabe verschiedener Modellierungsoptionen in den Mittelpunkt. Hierzu sei vorab gesagt, dass die Usability einer Sprache sinnvoll nur anhand einer groß angelegten Nutzerstudie bewertet werden kann. Dies würde jedoch über den Rahmen dieser Arbeit hinausgehen, weswegen im Folgenden versucht wird, die Sprache anhand möglichst objektiver Kriterien zu bewerten.

Bereits in Abschnitt 2.3.1 wurden psychologische und pädagogische Studien zum intuitiven Verständnis verschiedener (Programmier-)Sprachen eingeführt – insbesondere auch für Nutzer, die selbst keinerlei Programmiererfahrung vorweisen können. Die Erkenntnis dieser Studien, Schlüsseloperationen auf ganzen Wörtern oder Sätzen wie im Alltagsgebrauch beruhen zu lassen, konnte teilweise umgesetzt werden. Die Hauptkomponenten Agenten, Regeln, Initialisierungen, Observer, Komplexe und Abbruchbedingungen für die Simulation werden allesamt mit einleuchtenden Schlüsselwörtern eingeführt,

welche die Komponente selbst oder ihre Funktion im Kern beschreiben. Bei Variablen ist das Schlüsselwort `var` zudem bekannte Konvention aus vielen weiteren Programmierstandards wie *Scala*[30] oder der *ECMAScript Spezifikation*[36], auf der unter anderem auch *Javascript* basiert, und somit insbesondere Anwendern aus dem Programmierumfeld wohl bekannt.
Ein mögliches Beispiel für ein minimales Modell in *Re.action* mit den für die meisten Beispiele genutzten Agententypen aus Abbildung 2.5 findet sich in Listing 4.1. In der Definition der möglichen internen

```
1    agent A: x, c(u,p)
2    agent B: y
3
4    complex(a: A, b: B){
5    rule bindAndSplit:  a.c(u)//0, b//0 <=> a.c(p)+b.y  @1.0, 1.0
6    rule underspec:     a.x+?            => a.x+?, b    @1.0, 1.0
7    }
```

Listing 4.1: Minimalbeispiel mit grundlegenden Komponenten in *Re.action*

Zustände einer Site bei Agentendefinitionen innerhalb der Modellsignatur wie in Zeile 1 des Listings werden die möglichen Zuständen mittels Kommata getrennt. Dies wurde so gehandhabt, da es sich um eine Aufzählung der verschiedenen Site-Zustände handelt. Bediente man sich hier etablierter Schreibweisen aus dem Programmierkontext, so könnte man diese Zustände ebenfalls über Trennstriche '`|`' trennen, die in Programmiersprachen in der Regel ein *logisches Oder* repräsentieren. Dies würde zusätzlich verdeutlichen, dass sich die Sites immer in nur jeweils einem Zustand gleichzeitig befinden können. Da dies jedoch Vorwissen über Programmiersprachen voraussetzt, welches bei Anwendern aus der Biochemiedomäne nicht zwingend vorhanden ist, wurde die Formulierung mittels Kommata gewählt.
Die konkrete Modellierung der Agenteninstanzen und ihrer Verhältnisse untereinander weist keinerlei Schlüsselwörter mehr auf, sondern basiert nur auf Formulierungen, die mittels diverser Operatoren aufgebaut sind. Dies stellt erstmal einen großen Verlust für das intuitive Verständnis dar, da man sich von der Syntax gewohnter Alltagssprache auf die Syntax der Modellierung und deren eigene Semantik umstellen muss. Um diesen Verlust auszugleichen, werden zum einen Operatoren genutzt, die naturgemäß starke semantische Assoziationen wecken, wie zum Beispiel der '`+`'-Operator für Bindungen oder der '`//`'-Operator in der Regel `bindAndSplit` aus Zeile 5 zur Aufspaltung beziehungsweise Separation von Agenten und deren Verbindungen. Diese Operatoren basieren jedoch auch teilweise auf Dogmen verschiedener Programmiersprachen, wie der Zugriff auf Datenfelder oder Methoden eines Objekt in der objektorientierten Programmierung über einen Punkt '`.`' als Operator. Dies mag Nutzern, die aus einem Programmierumfeld stammen, sinnvoll erscheinen und auch intuitiv einleuchten. Nutzern aus der Biochemiedomäne könnte es jedoch zunächst willkürlich anmuten.
Die Wildcards '`+?`' und '`//0`' wie in den Zeilen 5 und 6 des Listings 4.1 führen die Idee der normalen Bindungsoperatoren auf eine neue Abstraktionsebene und die Kombination aus '`+`' und '`?`' als „*verbunden mit irgendetwas*“ ist quasi selbsterklärend. Die Null '`0`' als Platzhalter für ein „Nichts“ macht auch intuitiv Sinn. Durch den Zusammenhang mit dem Operator '`//`' könnte eine akribische, buchstäbliche Interpretation jedoch zu Missverständnissen führen, da es als „*nicht verbunden mit nichts*“ aufgefasst werden könnte, was in diesem Fall die falsche Semantik abbildet. Da *Re.action* durch die regeleigenen Signaturen sehr verbos wäre, wurde diese Vergrößerung des Sprachumfangs über die Einführung von Komplexen wie von Zeile 4 bis Zeile 7 in Listing 4.1 abgefangen.
Zum anderen wird die fehlende Verwandtschaft zur Alltagssprache durch die Anlehnung von Reaktionsstrukturen an die konventielle Schreibweise chemischer Reaktionen ausgeglichen. Dieser Grundsatz

wurde weitestgehend eingehalten und noch etwas erweitert. Diese Erweiterungen sind konkret die Möglichkeit bidirektionaler Reaktionsregeln und das Hinzufügen von Reaktionsraten. An manchen Stellen wurden diese Konventionen jedoch auch gebrochen. So werden die verschiedenen beteiligten Agenten in den Vor- und Nachbedingungen der Regeln mittels Kommata voneinander getrennt, wohingegen in den klassischen chemischen Reaktionsformeln alle beteiligten Reaktanden jeweils über ein Plus getrennt werden.
Zudem wurde die Anzahl an nötigen Klammerungen so gering wie möglich gehalten, da diese zwar gute Mittel zur Strukturierung von Sprachen darstellen, in großen Mengen aber den Lese- und Schreibfluss der Sprache stören. Klammern werden nur zur Definition von Site-Zuständen und Signaturen, in arithmetischen Ausdrücken oder in geschweifter Form zur Strukturierung von Komplexen benötigt. Bei der regelinternen Modellierung spielen sie also eine eher untergeordnete Rolle.

4.1.2. Gegenüberstellung mit anderen Sprachen aus dem Biochemiekontext

Um einen sinnvollen Vergleich zwischen verschiedenen Sprachen ziehen zu können, werden nun Regeln unterschiedlichen Umfangs den beiden anderen bereits in Abschnitt 2.3.1 eingeführten Spezifikationen *Kappa* und *BNGL* gegenüber gestellt.
Der Umfang weiterer Komponenten wie Agentendeklarationen in dafür vorgesehenen Modellsignaturen oder die Formulierung von Anfangsbedingungen werden zunächst vernachlässigt, da diese nur einmalig und normalerweise nicht maßgeblich zum Umfang des Modells beitragen. Ebenso werden bidirektionale Regeln nicht berücksichtigt, da diese analog in allen betrachteten Sprachen verfügbar sind.
Hierzu wird zunächst das Beispiel einer einfachen Bindung aus Abbildung 2.5 in Abschnitt 2.3.1 mit und ohne Zustandsänderung der Site `c` betrachtet. Die in den verschiedenen Sprachen formulierten Regeln zeigen Listing 4.2 und 4.3. Dort befinden sich in den Zeilen 1 und 2 jeweils Beispiele in *Re.action*

```
1  rule a_b (a: A, b: B):  a//0, b//0                  => a.x+b.y, a.c//0          @1.0
2  rule a_b:               a//0, b//0                  => a.x+b.y, a.c//0          @1.0
3  a_b:                    A(x, c) + B(y)              -> A(x!1, c).B(y!1)          1.0
4  'a_b'                   A(x[.], c[.]), B(y[.])      -> A(x[1], c[.]), B(y[1])   @1.0
```

Listing 4.2: Regel ‘a_b’ ohne Zustandsänderung

```
1  rule a_b (a: A, b: B):  a.x//0, a.c(u)//0, b//0   => a.x+b.y, a.c(p)//0          @1.0
2  rule a_b:               a.x//0, a.c(u)//0, b//0   => a.x+b.y, a.c(p)//0          @1.0
3  a_b:                    A(x, c~u) + B(y)          -> A(x!1, c~p).B(y!1)          1.0
4  'a_b'                   A(x[.], c[.]{u}), B(y[.]) -> A(x[1], c[.]{p}), B(y[1])   @1.0
```

Listing 4.3: Regel ‘a_b’ mit Zustandsänderung

(einmal mit und einmal ohne Signatur unter der Annahme, dass die Regel in einem passenden Komplex definiert wurde), in Zeile 3 jeweils die entsprechende Regel in der *BioNetGenLanguage* und in den Zeilen 4 wurde die Regel mittels *Kappa* spezifiziert.
Es wird auf den ersten Blick deutlich, dass der Regelkopf – bestehend aus Schlüsselwort, Name und Signatur – in *Re.action* deutlich mehr Platz einnimmt als in den anderen Sprachen. Diese benötigen weder ein Schlüsselwort noch eine Signatur und sind dadurch deutlich reduzierter. Aus diesem Grund wurden in *Re.action* Komplexe eingeführt, die die benötigte Eingabe im Regelkopf reduzieren. Dennoch fällt dieser in *Re.action* aufgrund des Schlüsselworts `rule` in der Regel größer aus und die Formulierung eines entsprechenden Komplexes benötigt selbst wieder Platz, was den Umfang des gesamten Codes

erhöht. Gleichzeitig können Komplexe wiederum als strukturierendes Element zur Zusammenfassung von Regeln gesehen zu werden.
Des Weiteren ist eine Signatur zwar eine weitere, Platz benötigende Komponente, die definiert werden muss, andererseits stellte sie sich jedoch auch als gutes Werkzeug zur Selbstüberprüfung heraus. Das vorherige Bewusstmachen und Festlegen der benötigten Agenteninstanzen in der Signatur erlaubt es, bereits bei und nach Eingabe der Regel im Editor darauf hinzuweisen, falls Agenteninstanzen zwar in der Signatur definiert wurden, aber in der Regel selbst ungenutzt blieben. Dies ist insbesondere nützlich, um menschliches Versagen bei der manuellen Eingabe von Regeln zu vermeiden, wenn zum Beispiel aus Versehen manche Agenteninstanzen in der Regel schlicht zu formulieren vergessen wurden.
Die Formulierung der Vorbedingung aus Listing 4.2 ist in *Re.action* deutlich kompakter als in den anderen Sprachen. Dies wird durch die kompakte Spezifikation aller Sites eines Agenten mittels des auf die Agenteninstanz angewendeten Operators '`//0`' möglich, wo in den anderen Sprachen alle freien Sites zumindest in irgendeiner Art und Weise erwähnt werden müssen.
Dieser Vorteil geht allerdings verloren, sobald es darum geht, die Sites wie in der Nachbedingung genauer zu definieren oder wenn wie in Listing 4.3 der interne Zustand einer Site definiert werden muss, obwohl sich die Site in einem freien Bindungszustand befindet. Hierbei bleiben *BNGL* und *Kappa* kompakter, da durch die Klammer- und indexbasierte Schreibweise jeder Agent nur einmal aufgerufen werden muss. In *Re.action* erfordert jeder Zugriff auf eine Site den vorherigen Aufruf der zugehörigen Agenteninstanz. Dies wird insbesondere dann zum Nachteil, wenn viele verschiedene Sites eines Agenten genauer definiert werden müssen – eine Ausnahme bildet hierbei nur die Definition aller Sites als freie Sites.
Allgemein ist die Kennzeichnung einzelner Sites als frei in *BNGL* durch die reine Erwähnung der Site in der Vor- oder Nachbedingung am kompaktesten gelöst. Es kommt hierbei komplett ohne zusätzliche Zeichen aus, wo *Kappa* und *Re.action* jeweils drei Zeichen für '`[.]`' oder '`//0`' benötigen.
Die Zuweisung der Reaktionsraten ist ausgewogen, da hier bei allen Regeln entweder einfach ein numerischer Wert, eine Variable oder ein allgemeiner arithmetischer Ausdruck aufgeführt werden kann. Hierbei unterstützen alle drei Sprachen den Einsatz von Dezimalzahlen und der wissenschaftlichen Schreibweise. Als nächstes muss man die Sonderfälle zur Erstellung oder Löschung von Agenten betrachten. Die Synthese und Degradation von Agenten ist als einfache bidirektionale Regel in Listing 4.4 gezeigt. In allen von nun an folgenden Beispielen ist die Signatur der in *Re.action* formulierten Regeln ausgelassen, da angenommen werden kann, dass die Regel in einem entsprechenden Komplex definiert ist.

```
1  rule synthDeg:  _       <=> a//0, b//0          @1.0, 1.0   #  Re.action
2  synthDeg:       0       <-> A(x, c~u), b(y)      1.0, 1.0   // BNGL
3  'synthDeg'      ., .    <-> A(), B()            @1.0, 1.0   // Kappa
```

Listing 4.4: Regel zur Agentensynthese

An der Bidirektionalität der Regeln wird deutlich, dass Synthese und Degradation in allen Sprachen analog definiert werden, nur mit Austausch der jeweiligen Vor- und Nachbedingung. Hierbei gibt es bei *Re.action* in Zeile 1 und *BNGL* in Zeile 2 einen einzigen Platzhalter für eine leere Bedingung. Die Löschung und Erstellung von Agenten in Regeln, die noch weitere Agenten beinhalten, welche allerdings nicht gelöscht und auch nicht erst neu erzeugt werden sollen, erfolgt ganz simpel dadurch, dass die entsprechenden Agenten nur auf der Seite der Regel erwähnt werden, auf der sie existieren sollen. Gelöschte Agenten tauchen also nur in der Vorbedingung und neu erzeugte Agenten nur in der Nachbedingung auf.
Kappa in Zeile 3 geht hier andere Wege, indem jeder Agent in der Auflistung aller Reaktanden/Produkte

eine feste Position besitzt. Wird ein Agent gelöscht oder neu erzeugt, muss diese Position mit dem Platzhalter für einen fehlenden Agenten, einem Punkt '`.`', gekennzeichnet werden. Hierbei offenbart sich eine Schwäche von *Kappa*, die es sehr fehleranfällig gemacht und exemplarisch in Listing 4.5 gezeigt ist. Die dort gezeigte Regel löscht den Agenten A der linken Seite und **auch** den Agenten B und ersetzt diesen Agenten B durch einen neuen Agenten A mit den in der Nachbedingung der Regel aufgeführten Eigenschaften. Allerdings könnte man die Regel auch so auslegen, dass nur der Agent B gelöscht wird, da der Agent A auf beiden Seiten vorhanden ist. Eine eindeutige Zuweisung von Positionen wie in *Kappa* gibt es in *Re.action* nicht und ist auch gar nicht notwendig, da die verschiedenen Agenteninstanzen dort über ihre Namen eindeutig identifiziert werden. Daher sind eigene Reaktionssignaturen in *Re.action* zur namentlichen Deklaration von Agenteninstanzen zwingend notwendig. Somit ist auch die Reihenfolge, in der die Bindungen erscheinen, dort nicht von Belang, was die Fehleranfälligkeit in solchen Regeln deutlich verringert, da es viel unwahrscheinlicher ist, dass aus Versehen (falsche) Agenten gelöscht oder erzeugt werden.

```
'synthDeg'      A(x[.], c[.]), B(y[.]) -> ., A(x[.], c[.])  @1.0
```

Listing 4.5: Fehleranfällige Regel in *Kappa*

Eine weitere zu vergleichende Komponente der verschiedenen Spezifikationen sind *Observer* bzw. *Observables*, wie sie in *BNGL* und *Kappa* genannt werden. Beispieldefinitionen dieser Observer zeigt Listing 4.6. Dabei muss das Observable aus Zeile 2 in *BNGL* innerhalb einer '`begin observables`

```
1 observe     Acp:    a.c(p)//0       #  Re.action
2 Molecules   Acp     A(c~p)          // BNGL
3 %obs:       'Acp'   |A(c[.]{p})|    // Kappa
```

Listing 4.6: Observer/Observables in verschiedenen Spezifikationen

`<...> end observables`'-Umgebung definiert sein. Um den Fokus auf der Observable-Definition zu halten, wird an dieser Stelle davon ausgegangen, dass dies der Fall ist.
Im Allgemeinen sind sich die Spezifikationen auch hier relativ ähnlich, jedoch werden die Komponenten in *BNGL* (Zeile 2) und *Re.action* (Zeile 1) sehr lakonisch formuliert, wohingegen das *Observable* in *Kappa* (Zeile 3) viele weitere Zeichen benötigt, die die Eingabe umständlich gestalten und wenig Nutzen für die Definition der Komponenten selbst bieten. Das Prozentzeichen '`%`' zu Beginn, sowie die Hochkommata '`'`', die den Namen umgeben, und die Striche '`|`' um das zu beobachtende Muster herum, sind allesamt Sonderzeichen, die nicht ganz einfach auf den meisten Tastaturen einzugeben sind, und den gesamten Ausdruck sehr verbos machen. Generell ist die Schreibweise von Mustern in *Kappa* durch die hohe Zahl verschiedener Klammern und genutzter Klammerarten eher umständlich, da sie ebenfalls Sonderzeichen sind, die auf den meisten Tastaturen Tastenkombinationen und somit mehr als einen einfachen Tastendruck wie ein Punkt benötigen. Hier schneiden die anderen beiden Sprachen deutlich besser ab.
Ein weiterer, in biochemischen Systemen häufig formulierter Fall ist zum Beispiel die Auflösung einer einzelnen Bindung. Eine solche Regel in den verschiedenen Sprachen ist in Listing 4.7 gezeigt.

```
1  rule split:     a.x+b.c          => a.x//b.c        @1.0    #  Re.action
2  split:          A(x!1).B(y!1)    -> A(x) + B(y)      1.0    // BNGL
3  'split'         A(x[1/.]), B(y[1/.])                @1.0    // Kappa
```

Listing 4.7: Regel zur Auflösung einer einzelnen Bindung

Die verkürzende Schreibweise von *Kappa* in Zeile 3 fällt direkt auf, da eine explizite Formulierung von Vor- und Nachbedingung komplett entfällt. Diese werden implizit durch den Ausdruck '`[1/.]`' im Index der Sites definiert. Diese Schreibweise macht die gesamte Regel unglaublich kompakt, auch wenn sie sich dafür von der konventionellen Schreibweise chemischer Reaktionen entfernt. In *Re.action* aus Zeile 1 und *BNGL* aus Zeile 2 fallen diese durch die explizite Formulierung der Vor- und Nachbedingung deutlich umfangreicher aus. Hierbei ist *Re.action* insbesondere ineffizient, wenn die Regel bidirektional formuliert sein soll, wie die Nachbedingung in Listing 4.8 zeigt. Damit die modellierte Regel ihre

```
1  rule splitBi:   a.x+b.c          <=> a.x//0, b.c//0  @1.0, 1.0
```

Listing 4.8: Bidirektionale Regel zur Auflösung einer einzelnen Bindung in Re.action

Korrektheit behält, müssen die Sites in der ursprünglichen Nachbedingung nun konkret als frei und nicht nur als „*nicht miteinander verbunden*“ definiert werden, da die vorherige Nachbedingung nun zugleich die Vorbedingung der Regel in Rückrichtung beschreibt. Würde man es hier bei der alten Schreibweise belassen, wäre zwar definiert, dass die Sites `x` und `c` nicht miteinander verbunden sind, jedoch sind sie nicht zwangsweise frei, da sie dennoch mit beliebigen anderen Sites (anderer Agenten) verbunden sein könnten.
Bei sehr großen Verbindungen wie in der Nachbedingung der Regel in Listing 4.9 nehmen die verschiedenen Sprachen ungefähr gleich viel Platz ein. Hierbei stellen die Bindungen aus Zeile 1 in *Re.action* deutlich in den Vordergrund, welche beiden Sites miteinander verbunden sind, da dies direkt über einen zweiwertigen Operator '`//`' oder '`+`' definiert wird. In Kontrast dazu müssen bei *BNGL* bzw. *Kappa* in den Zeilen 2 bzw. 3 durch die indexbasierte Schreibweise erstmal die zugehörigen Sites im Muster gefunden werden. Dieses „Finden“ zusammengehöriger Sites tritt jedoch auch in *Re.action* auf, sobald es um Verbindungen mit mehr als einem Agenten geht. Hierbei müssen dann zwar nicht die zusammengehörigen Sites, sondern weitere Erwähnungen derselben Agenteninstanz gefunden werden (z.B. `a2`, welche in zwei Bindungen vorkommt).
Zudem gilt es auch, den Funktionsumfang der verschiedenen Spezifikationen zu beachten. Allein bei

```
1  rule bigPattern:    a1//0, a2//0, b//0
2                              => a1.x+a2.c, a2.x+b.y, a1.c//0
3
4  bigPattern:         A(x, c) + A(x, c) + B(y)
5                              -> A(x!1, c).A(x!2, c!1).B(y!2)
6
7  'bigPattern'        A(x[.], c[.]), A(x[.], c[.]), B(y[.])
8                              -> A(x[1], c[.]), A(x[2], c[1]), B(y[2])
```

Listing 4.9: Regel mit großer Nachbedingung (Reaktionsraten aus Platzgründen ausgelassen)

den Modellierungsoptionen innerhalb von Regeln bieten *BNGL* und *Kappa* auch die Auflösung unterspezifizierter Bindungen an, wie es beispielsweise in Listing 4.10 gezeigt ist. Diese Art von Reaktionen sind

in *Re.action* (Zeile 1) verboten, da es mit dem zugrunde liegenden stark typisierten Simulationsmodell nicht möglich ist, die Bindung zu einem unbekannten Agenten auflösen. Dies macht die anderen beiden Sprachen in den Zeilen 2 und 3 für Anwendungsfälle, wo dies vonnöten ist, ungleich ausdrucksstärker. In *Re.action* müsste für diese eine Regel ein ganzes Set an Regeln erstellt werden, die alle möglichen Bindungspartner der Site abdecken.

```
1 rule underspec (a: A):  a.x+?   => a.x//0   @1.0    #  Re.action: verboten
2 underspec:              A(x!?)  => A(x)      1.0    // BNGL:      erlaubt
3 'underspec'             A(x[_]) -> A(x[.])  @1.0    // Kappa:     erlaubt
```

Listing 4.10: Unterspezifizierte Bindungsauflösung in *Re.action* und *Kappa*

4.1.3. Evaluationsergebnisse

Es zeigt sich, dass alle untersuchten Sprachen ihre Stärken und Schwächen haben, die in verschiedenen Modellierungsarten zum Vorschein kommen. Wo *Re.action* beispielsweise komplett freie Agenten kompakt definieren kann, schafft dies *Kappa* durch Kurzschreibweisen zur Auflösung konkreter Verbindungen oder *BNGL* durch die insgesamt kürzeste Schreibweise zur Definition freier Sites.
Die für *Re.action* gesetzten Ziele konnten in nicht allen Punkten erreicht werden, jedoch gibt es für die meisten Ausdrücke eine kompakte Schreibweise, die im Regelfall kürzer oder zumindest ungefähr gleich viel Platz einnehmen wie die gleichen Konstrukte in anderen Sprachen.
Durch die einzelnen Reaktionssignaturen bietet es den Nutzern ein Feature zur Selbstkontrolle, wie es in den anderen Sprache nicht existiert. Der Aufbau von Mustern als Summe zweiwertiger Bindungen, welche über die Operatoren '`//`' und '`+`' definiert werden, ist direkt klar und nachvollziehbar, wird jedoch bei Verbindungen aus mehreren Agenten eventuell unübersichtlich. Welche Schreibweise angenehmer zu formulieren und zu lesen ist, hängt allerdings stark von den persönlichen Präferenzen des Anwenders ab. Hierbei bietet *Re.action* zumindest eine Alternative zur Schreibweise der beiden aktuellsten Vertreter *BNGL* und *Kappa*, die beide indexbasiert arbeiten.
Allerdings bietet *Re.action* nicht die Modifikation unterspezifizierter Verbindungen an. Dies kann durch eine Menge passender Regeln ersetzt werden, die alle möglichen Bindungspartner der unterspezifizierten Site abdecken, ist in den beiden anderen Sprachen jedoch deutlich kompakter. In der Praxis wird dieser Fall allerdings selten benötigt. In dem (durchaus umfangreichen) Modell von *Proctor und Gray* (s. Appendix D) wird dies zum Beispiel in keinem Muster benötigt.
Zudem werden in dieser Evaluation nur die Aspekte der Sprachen zur Modellierung des biochemischen Systems selbst betrachtet. Der Umfang an Optionen zur Steuerung der Simulation und der von ihr genutzten Methode ist in *BNGL* und *Kappa* deutlich größer, da sie schon viel länger vorhandene und somit auch ausgereiftere und erweiterte Simulatoren besitzen, als es mit *SimSG* der Fall ist.
Die Frage, welche Sprache man unabhängig vom damit verbundenen Simulationstool für bestimmte Modellierungen nutzen soll, lässt sich allgemein nicht beantworten, da es hierbei stark auf die Art und Weise, wie ein System modelliert wurde, und die dadurch resultierende Menge an Regeln ankommt. Falls man schon ein vollständiges Modell vorliegen hat, sollte man dessen Regeln auf die während der Evaluation hervorgehobenen Umstände, unter denen verschiedene Sprachen ihre Vorteile zur Geltung bringen, untersuchen und auf dieser Basis eine Entscheidung treffen.

4.2. Evaluation verschiedener Mustererkennungstools

Da mit der entwickelten *Re.action* DSL ein Tool geschaffen wurde, um schnell und einfach neue Modelle zu erstellen, bietet es sich an, verschiedene Pattern-Matching-Engines anhand diverser Modelle gegeneinander zu evaluieren.
Dieses Kapitel widmet sich der Evaluation der in *eMoflon* vorhandenen Pattern Matcher *Democles* und *HiPE*. Die Hintergründe zu diesen beiden Erkennungstools wurden bereits in Abschnitt 2.3.3 erläutert. Des Weiteren geht es darum, den Einfluss der in Abschnitt 3.4.3 dargelegten Änderung der unidirektionalen Kantenrealisierung im Simulationsmodell zur bidirektionalen Kantenstruktur zu ermitteln. Dazu werden schließlich zu allen genutzten Evaluationsmodellen Simulationsmodelle beider Varianten generiert und getestet.
Die bei der Evaluation betrachteten Modelle werden in Abschnitt 4.2.1 näher beleuchtet und Abschnitt 4.2.2 gibt zusammenfassend einen Überblick über die Evaluationsergebnisse.

4.2.1. Evaluationsaufbau

Zur Evaluation der Pattern Matcher wird ein Intel® Core™ i7-6700HQ mit 4 Kernen und 8 Threads genutzt, die mit einer Frequenz von jeweils circa 3.10 GHz betrieben werden. Als RAM werden 16GB DDR3 Arbeitsspeicher zur Verfügung gestellt.
Für die verschiedenen Benchmarks wurden die von *SimSG* ausgegebenen Daten zum während der Simulation verbrauchten Speicher und der Laufzeit der Simulation genutzt. Alle Simulationen wurden aufgrund der teilweise hohen Laufzeiten nur drei Mal durchgeführt. Somit sind statistische Abweichungen in den meisten Plots vermutlich nicht komplett erfasst. Sollten in einem Plot keine Fehlerbalken dargestellt sein, handelt es sich bei den dargestellten Werten um den Median der Datenreihe.

In den Evaluationssimulationen werden drei unterschiedliche biochemische Modelle genutzt. Das erste Modell ist die sogenannte *Goldbeter-Koshland-Loop* (GKL) [40], die ein Beispiel mit großer Praxisrelevanz darstellt, da sie häufig in verschiedenen biochemischen Systemen auftaucht. Sie ist zugleich ein Beispiel für ein relativ simples Modell, da sie nur drei Agententypen K, T und P besitzt, und wurde bereits 2007 von *Danos et al.* genutzt [8]. Die Agenten K und P besitzen jeweils nur eine Site a ohne interne Zustände. Der Agent T verfügt über zwei Sites x und y, welche jeweils die Zustände u und p besitzen. Die Zustände beschreiben jeweils, ob sich die Site in unphosphoryliertem oder phosphoryliertem Zustand befindet.
Agenten des Typs K und P verbinden bzw. lösen sich nun ständig mit bzw. von Sites an Agenten des Typs T. Über welche Sites von T die Bindung stattfindet, ist hierbei nicht von Belang, da die weitere Funktionsweise für die Sites x und y die gleiche ist, und bei den Agenten K und P gibt es nur eine Site, über die Verbindungen hergestellt werden können.
Die Agenten K wirken nun als sogenannte *Kinase*, wodurch sie unphosphorylierte Sites, mit denen sie verbunden sind, phosphorylieren. Konkret wird der Zustand der entsprechenden Site also von u zu p geändert.
Bei Bindungen von Agenten T mit Agenten vom Typ P läuft dieser Prozess vice versa ab, sodass P als sogenannte *Phosphatase* wirkt, die die phosphorylierten Sites von T wieder dephosphoryliert - also in den Zustand u zurückversetzt.
Hierbei werden während der Simulation alle Agenten T beobachtet, bei denen alle Sites phosphoryliert sind, und zugleich die Teilmenge all dieser Agenten, bei denen beide Sites frei sind.
Zu Beginn der Simulation werden Instanzen freier und komplett unphosphorylierter Agenten der Typen

K, T und P erzeugt. Die Anzahl wird über verschiedene Modellgrößen zu je 100, 200, 400, 800 und 1600 generierten Instanzen skaliert. Die Anzahl der Instanzen ist äquivalent zu der Anzahl der Knoten im Graphen, auf dem die Pattern Matcher operieren, da jede Instanz durch einen eigenen Knoten repräsentiert wird. Alle Simulationen sind hierbei auf 30000 Iterationen begrenzt. Das entsprechende in *Re.action* formulierte Modell befindet sich in Appendix C.
Als zweites Modell wird das in Abschnitt 2.1.1 eingeführte Modell von *Proctor und Gray* [9] zur Untersuchung des Einflusses der Enzyme *GSK3*β und *Mdm2* mit Proteasomen auf die Entwicklung von Alzheimer. Da das Modell dort schon zur Genüge beschrieben wurde und diese Arbeit nicht genug Raum bietet, um wirklich detailliert auf ein solch komplexes Modell einzugehen, wird es hier nicht weiter erläutert.
In der Arbeit von *Proctor und Gray* wurden sowohl Simulationen unter Normalbedingungen als auch unter Belastung durch Infrarotstrahlung, die als eigener Agententyp modelliert wurde und mit einer bestimmten Instanzmenge nach einem gewissen Zeitraum in das Modell eingespeist wurde, durchgeführt. Da *SimSG* das Hinzufügen von neuen Instanzen in das System abseits von Anfangsbedingungen nicht unterstützt, wird zur Evaluation hier nur das Modell unter Normalbedingungen genutzt. Die Ergebnisse dieses Modells stehen in direktem Kontrast zur Goldbeter-Koshland-Loop, da es ungemein mehr Agententypen und Regeln enthält. Das gesamte in *Re.action* formulierte Modell findet sich in Appendix D.
Da die originalen Anfangsbedingungen aus der Studie ein Modell mit über 18000 Knoten generieren, das von dem Testgerät nicht simuliert werden kann, da es zu einem Heap-Overflow führt, wurden diese Anfangsbedingungen um den Faktor 5 herunter skaliert. Dadurch geht selbstverständlich die biochemische Genauigkeit der Simulation verloren, allerdings reicht dies lediglich zur Evaluation der Performance dennoch aus.
Ein Grund für den intensiven Ressourcenbedarf der Simulation ist vermutlich auch die hohe Zahl an disjunkten Teilmustern im Reaktionsmodell. Solche disjunkten Teilmuster sind Muster, deren Agenten nicht mit den anderen Agenten des Musters verbunden sind. Somit lässt sich das gesamte Muster in mehrere unverbundene Subgraphen aufteilen, welche eben diese genannten disjunkten Teilmuster darstellen. Da die Agenten nicht explizit miteinander verbunden sind, müssen beispielsweise für ein Muster bestehend aus drei freien Agenten X, Y und Z alle möglichen Kombinationen aus Knoten diesen Typs gematcht werden. Dieses Muster besitzt nun 3 dieser disjunkten Teilmuster, die jeweils aus einem freien Knoten des Typs X, Y oder Z bestehen. Liegt ein Modell mit jeweils 5 freien Instanzen dieser Agenten vor, so findet der Pattern Matcher 5 Matches für jedes dieser Teilmuster. Als Match des gesamten Musters muss also das wie in Formel 4.1 definierte kartesische Produkt der Mengen aller Matches der Teilmuster gebildet werden. Das auf n Mengen erweiterte kartesische Produkt ist in Formel 4.2 definiert.

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\} \tag{4.1}$$

$$X_1 \times \cdots \times X_n = \{(x_1, \ldots, x_n) \mid x_i \in X_i, \text{ für } \forall i \in \{1, \ldots, n\}\} \tag{4.2}$$

Somit ergibt sich die Zahl der Matches des tatsächlichen Musters aus $|X \times Y \times Z| = |X| \cdot |Y| \cdot |Z| = 5 \cdot 5 \cdot 5 = 125$. Diese große Zahl an potenziellen Matches kann bei vielen disjunkten Teilmustern in Regeln schnell zu einer kombinatorischen Explosion der gefundenen Matches führen, was den Simulationsprozess sehr rechenintensiv macht.
Da in dem Modell von *Proctor und Gray* viele Agenten vorkommen, die gar keine Sites besitzen, wie zum Beispiel der Energieträger ***A**denosin**t**ri**p**hosphat* (ATP), der im weitesten Sinne nur als Katalysator für Reaktionen dient, aber nie an andere Agenten bindet, treten solche disjunkten Teilmuster zuhauf in dem Modell auf. Aufgrund des hohen Ressourcenbedarfs werden die Simulationen hier schon nach

10000 Iterationen terminiert.
Zuletzt wird der Einfluss von verschieden großen Mustern auf die Performance der Pattern Matcher anhand eigens dafür konstruierter, simpler Modelle untersucht. Diese Modelle bestehen aus jeweils einer einzigen Regel, die eine bestimmte Menge von Agenten mit gleicher Rate aneinander bindet und wieder voneinander löst. Dazu werden im kleinsten Modell nur zwei Agenten als Musterparameter genutzt, wobei die Parameterzahl stets um einen Agenten inkrementiert wird, bis im größten Modell sechs Agenten innerhalb der Regel genutzt werden. Das kleinste und größte Modell sind exemplarisch in den Listings 4.11 und 4.12 gezeigt (die Reaktionssignaturen wurden der Übersicht halber entfernt, als würden sie sich in einem entsprechenden Komplex befinden). Als Anfangsbedingungen werden

```
1 agent U: a, b
2 agent V: a, b
3
4 init 20: u//0, v//0
5
6 rule osc: u.a//0, v.a//0 <=> u.a+v.a    @1.0, 1.0
7
8 terminate iterations = 1000
```

Listing 4.11: Modell und Regel mit zwei Parametern

```
1  agent U: a, b
2  agent V: a, b
3  agent W: a, b
4  agent X: a, b
5  agent Y: a, b
6  agent Z: a, b
7
8  init 20: u//0, v//0, w//0, x//0, y//0, z//0
9
10 rule osc:
11 u.a//0, v.a//0, w.a//0, x.a//0, y.a//0, z.a//0
12     <=> u.a+v.a, w.a+x.a, y.a+z.a   @1.0, 1.0
13
14 terminate iterations = 1000
```

Listing 4.12: Modell und Regel mit sechs Parametern

jeweils 20 Instanzen aller an der Regel beteiligten Agenten erstellt und alle Simulationen führen 1000 Iterationen aus. Da die Regel in den hier genutzten Modellen auf einer Seite aus jedem Parameter als freiem Agent besteht, sind stets disjunkte Teilmuster vorhanden, die genau wie im Modell von *Proctor und Gray* auch ab einer gewissen Größe für einen massiven Mehrbedarf an Ressourcen bei steigender Parameterzahl sorgen werden.

Alle Evaluationen auf diesen Modellen werden sowohl mit dem in Abschnitt 3.4.3 beschriebenen unidirektionalen Metamodell als auch mit dem neuen, bidirektionalen Metamodell im Simulationsmodell ausgeführt. Es sei erneut darauf verwiesen, dass das unidirektionale Modell keine korrekte, vollständige Abbildung der biochemischen Reaktionen auf das Simulationsmodell darstellt. Es wird jedoch trotzdem verwendet, um die Auswirkungen der Änderungen im bidirektionalen Modell qualitativ und quantitativ bewerten zu können.

4.2.2. Evaluationsergebnisse

Goldbeter-Koshland-Loop

Das Simulationsergebnis des in Appendix C formulierten Modells mit 100 Instanzen jedes Agententyps als Anfangsbedingung liefert den Populationsverlauf in Abbildung 4.1. Unter „Population“ ist hierbei die Menge an Vorkommnissen des beobachteten Musters im Graphen zu verstehen. Hier also Knoten des Typs T mit zwei phosphorylierten Sites, deren Bindungsstatus einmal undefiniert und einmal explizit *frei* ist. Die entsprechende Simulation wurde mit der *Democles*-Engine und dem Metamodell mit bidirektionalen Kanten durchgeführt. Bei der zeitlichen Einheit handelt es sich hierbei um Sekunden.
Laufzeit und Speicherverbrauch der Simulationen über die skalierte Instanzen- bzw. Knotenzahl von

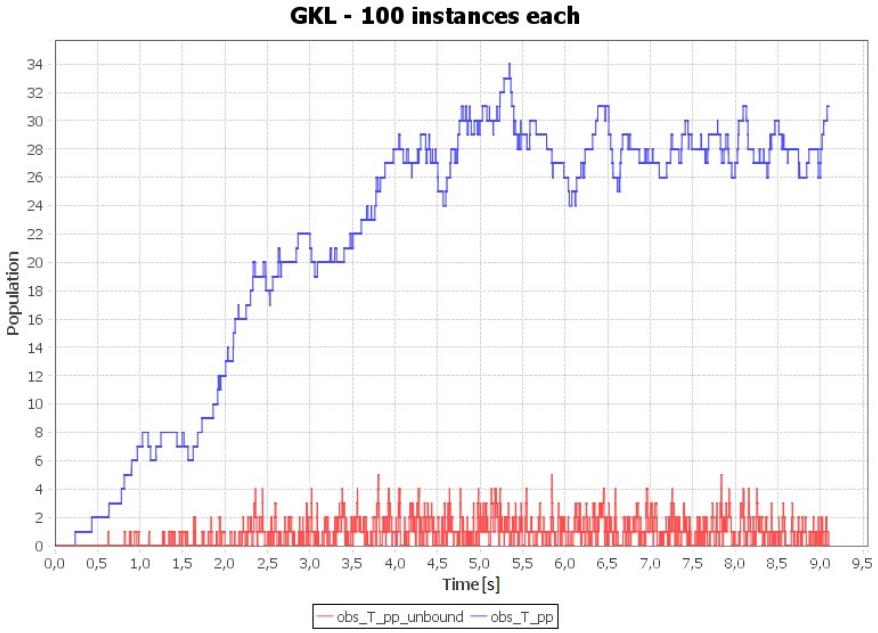
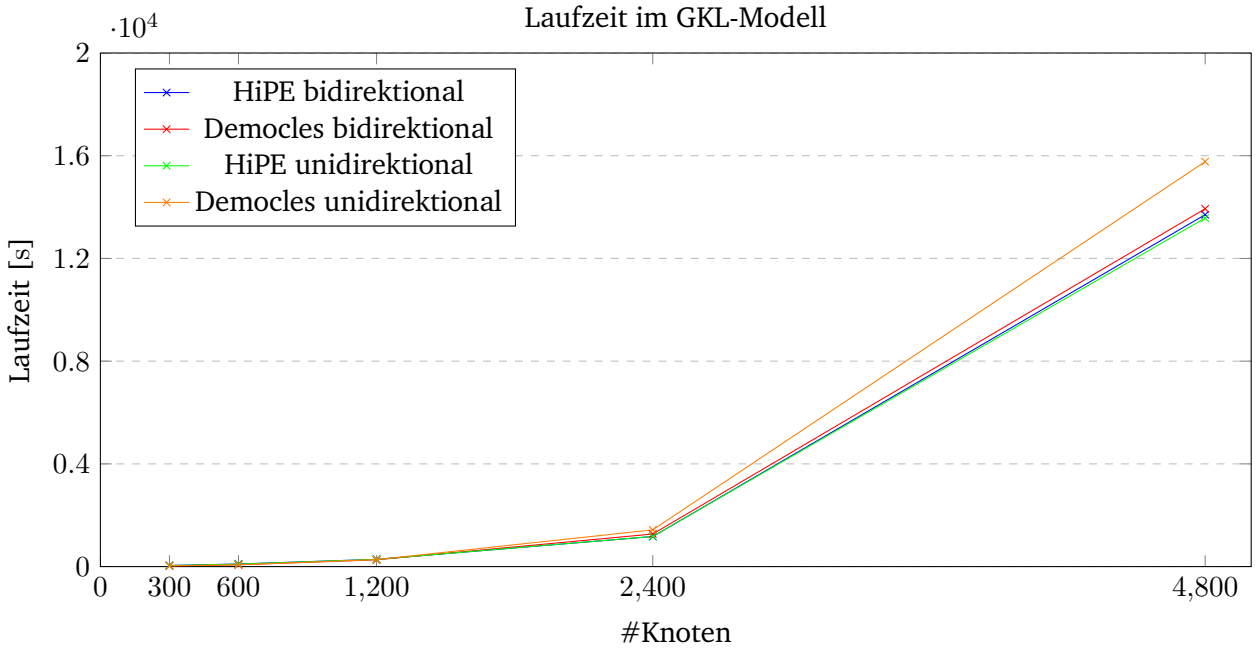


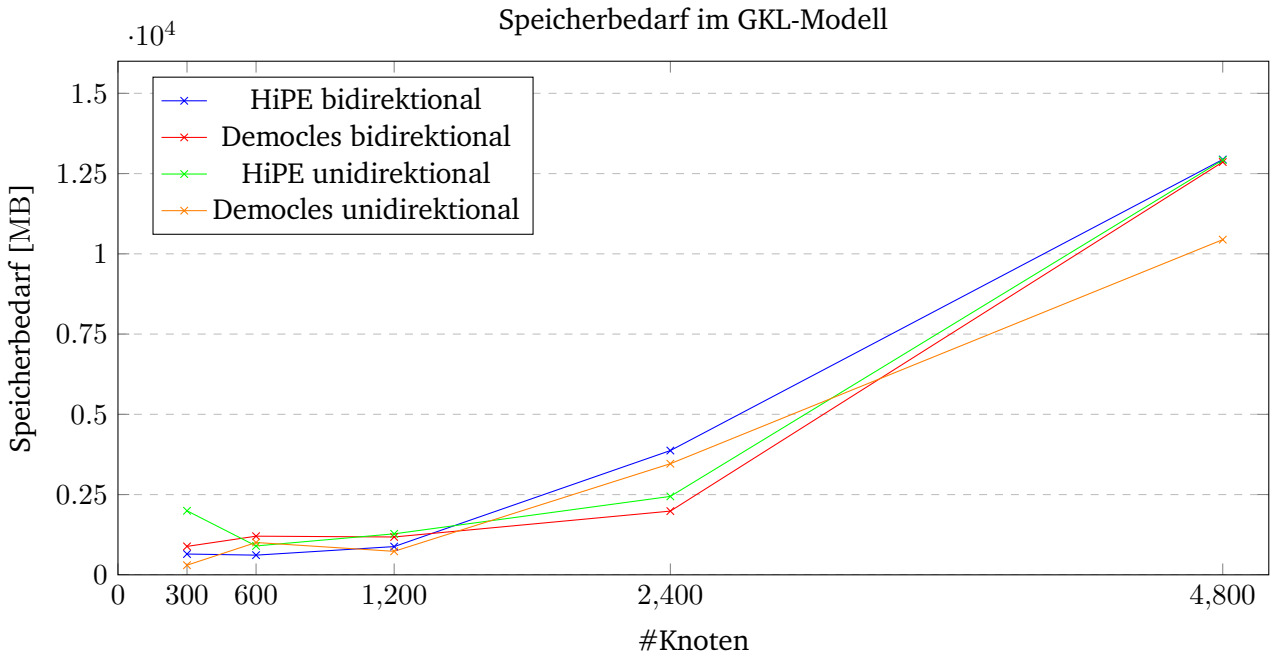
Abbildung 4.1.: Ergebnis der GKL-Simulation mit 300 Instanzen nach 30000 Iterationen

Democles und *HiPE* auf Basis eines Simulationsmetamodells mit unidirektionalen und mit bidirektionalen Kanten zeigt Abbildung 4.2. Die Darstellungen der Messdaten mit Fehlerbalken sind hierbei für eine übersichtliche Visualisierung in Abbildung 4.3 ausgelagert. Bei den Plots in Abbildung 4.2 handelt es sich um die Mediane der Datenreihen.
Das Laufzeitdiagramm in Abbildung 4.2a zeigt einen quasi-exponentiellen Anstieg für alle genutzten Kombinationen aus Pattern-Matching-Engine und Metamodell. Alle möglichen Konfigurationen aus Pattern-Matching-Engine und Kantenstruktur verhalten sich ungefähr gleich, jedoch benötigt *Democles* mit unidirektionalen Kanten etwas länger zur Ausführung der Simulation.
Das Diagramm zum Speicherbedarf in Abbildung 4.2b verhält sich zwar bis nicht so einheitlich wie die Laufzeit, allerdings lassen sich insbesondere aufgrund der statistischen Abweichungen (vgl. Abbildung 4.3b) nur schwer präzise Schlüsse über den Speicherbedarf der verschiedenen Konfigurationen ziehen. Es zeigt sich jedoch zumindest bei einer Knotenzahl von 4800, dass *Democles* mit unidirektionalen Kanten tendenziell weniger Speicher verbraucht als die anderen Konfigurationen.

Die Umstellung des Modells von bidirektionalen Kanten auf unidirektionale Kanten zeigt bei *HiPE* keinen

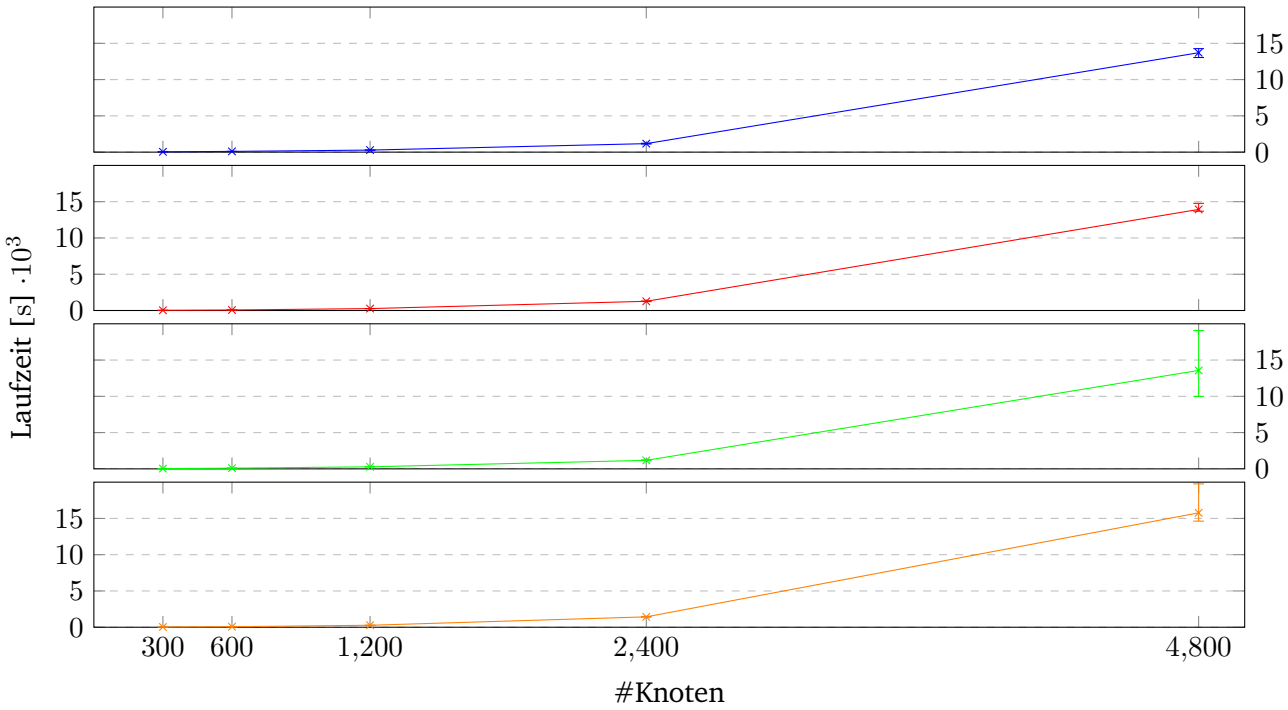


(a) Laufzeit in Abhängigkeit der Knotenzahl

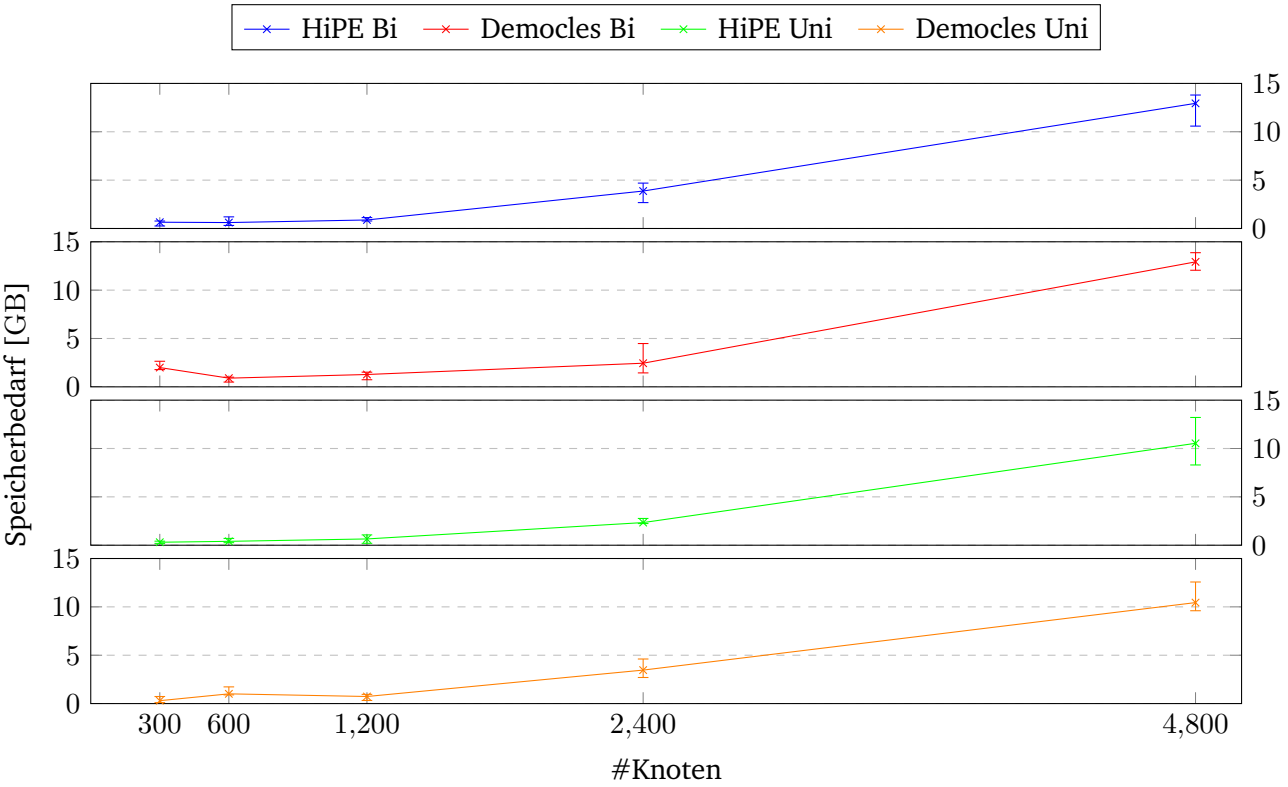


(b) Speicherbedarf in Abhängigkeit der Knotenzahl

Abbildung 4.2.: Evaluationsergebnisse der *Goldbeter-Koshland-Loop*



(a) Laufzeit im GKL-Modell mit Fehlerbalken



(b) Speicherbedarf im GKL-Modell mit Fehlerbalken

Abbildung 4.3.: Evaluationsergebnisse des GKL-Modells mit Fehlerbalken

Effekt, bei Democles führt es zwar zu einer Verbesserung der Laufzeit, jedoch auch einem Mehrbedarf an Speicher. Vergleicht man die beiden verschiedenen Engines innerhalb derselben Kantenkonfiguration miteinander, so ist *HiPE Democles* bei unidirektionalen Kanten in Bezug zur Laufzeit noch etwas überlegen. Dieser Nachteil scheint sich im Modell mit bidirektionalen Kanten jedoch aufzuheben.
Um die Ursache hierfür zu finden, sollte man einen genaueren Blick in die Unterschiede der *IBeX*-Patterns bei den verschiedenen Kantenstrukturen werfen, da diese das Verhalten der Pattern Matcher grundlegend bestimmen. Hierbei fällt neben der starken Typisierung der Kanten auf, dass beim Modell mit bidirektionalen Kanten viel mehr *Negative Application Conditions* (NACs) in Mustern mit freien Agenten enthalten sind. Die Ursache hierfür ist die starke Typisierung, wodurch mehr Referenzen im Metamodell entstehen, welche alle mittels NACs verboten werden müssen, um eine Site als *frei* definieren zu können. Im Modell mit unidirektionalen Kanten genügte eine einzelne NAC mit einer Kante, da diese durch ihre schwache Typisierung direkt alle möglichen Verbindungen zu anderen Agenten abdeckte. Zudem sind für jede Verbindung zwischen Agenten in einem *IBeX*-Pattern nur noch halb so viele Kanten vorhanden, die spezifiziert sein müssen, um Typsicherheit gewährleisten. Zusammengefasst ist die Anzahl an Teilmustern, die für ein Match gefunden werden müssen, geringer bzw. deren Umfang kleiner. Diese Änderungen reduzieren insbesondere bei sequenzieller Arbeitsweise den Arbeitsaufwand und damit die Laufzeit, wie es bei *Democles* der Fall ist. Beim parallel arbeitenden *HiPE* bleibt dies jedoch weitestgehend ohne Auswirkungen.

GSK3b-Mdm2-Modell

Bei der Simulation des Modells von *Proctor und Gray* [9] wurden 10000 Iterationen als Abbruchbedingung gesetzt und auf zwanzig Prozent reduzierte Anfangswerte gegenüber der ursprünglichen Werte aus der Studie genutzt. Das Ergebnis einer solchen Simulation mit bidirektionalen Kanten im Metamodell und mit *HiPE* als Pattern-Matching-Engine zeigt Abbildung 4.4. Die zeitliche Einheit beträgt auch hier Sekunden.

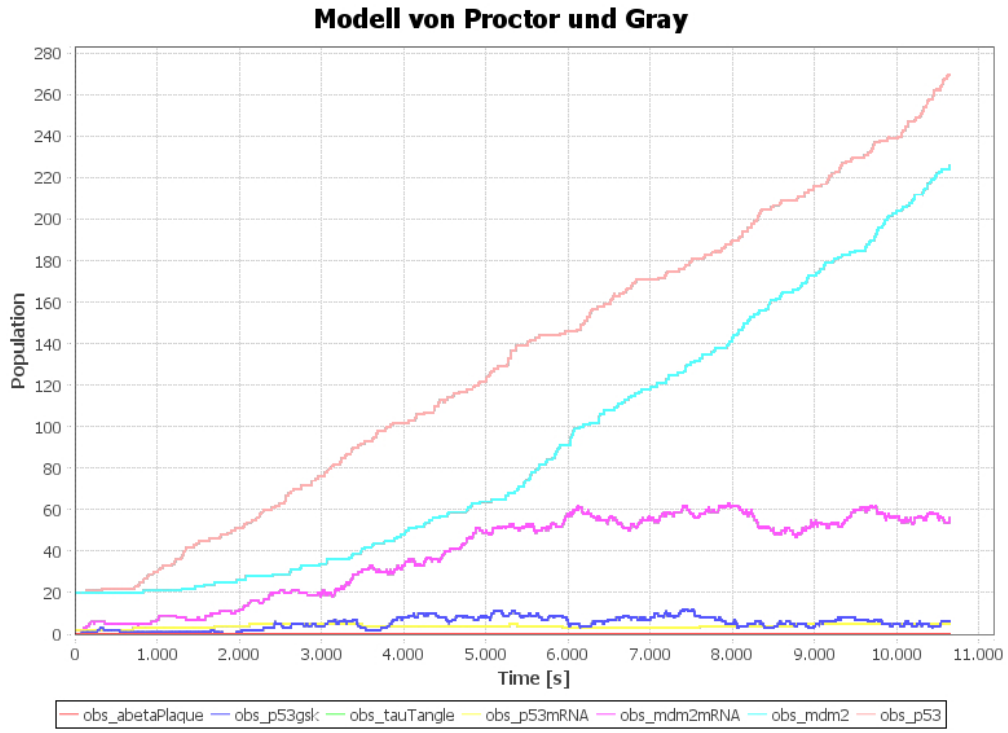


Abbildung 4.4.: Ergebnis der Simulation des Modells von *Proctor und Gray* mit 30000 Iterationen

Im Vergleich mit der Arbeit von *Proctor und Gray* fällt direkt auf, dass sich das Modell hier anders verhält. Dies lässt sich durch die Hinabskalierung der Anfangsbedingungen erklären, wodurch die Simulationsergebnisse im biochemischen Sinne nicht mehr korrekt sind. Für eine Bewertung der Laufzeit und des Speicherverbrauchs in einem praktischen und umfangreichen Beispiel reicht dies allerdings dennoch aus.
Die Evaluationsergebnisse für die verschiedenen Metamodelle und Pattern-Matcher zeigt Abbildung 4.5. Hierbei fällt direkt auf, dass *Democles* mit unidirektionalen Kanten im Metamodell eine außerordentlich erhöhte Laufzeit im Verhältnis zu den anderen Konfigurationen hat. Es zeigen sich also ähnliche Verhältnisse wie bereits bei den Ergebnissen der Goldbeter-Koshland-Loop, jedoch nimmt die Verbesserung der Laufzeit von *Democles* durch eine Umstellung auf bidirektionale Kanten im Metamodell ein extremeres Ausmaß an. Dies lässt sich auf die viel größere Anzahl an Regeln und Mustern im Modell von *Proctor und Gray* zurückführen. Die starken Schwankungen in der Ausführung dieser Konfiguration lassen sich dadurch begründen, dass die Anzahl an Agenten verschiedener Typen in den Anfangsbedingungen sehr heterogen verteilt ist. Es gibt manche Agenten, von denen nur eine einzelne Instanz existiert, während von anderen Agenten mehrere hundert oder sogar tausende Instanzen vorhanden sind. Wird durch die stochastische Auswahl der als nächstes auszuführenden Regel nun innerhalb einer einzelnen Simulation oft eine Regel gewählt, deren beteiligte Agenten in großen Mengen im System vorhanden

sind, so wird sich dies direkt deutlich auf die Laufzeit auswirken, da hier mit viel mehr Matches gearbeitet werden muss als bei Regeln mit Agententypen, von denen nur wenige im System enthalten sind. Bei *Democles* mit bidirektionalem Kantenmodell kann dieser Aufwand auf dieselbe Weise wie im GKL-Modell dieses Mal deutlich reduziert werden, indem die Anzahl der zu prüfenden Kanten und Agenten durch die starke Typisierung der Kanten schon in den *IBeX*-Patterns stark eingeschränkt wird und generell nur halb so viele Kanten in den Mustern enthalten sind, als es im unidirektionalen Modell der Fall war.
Im Vergleich zwischen *HiPE* und *Democles* innerhalb der jeweiligen Kantenkonfigurationen, schneidet *HiPE* bei unidirektionalen Kanten bei Weitem besser ab. Dieser Vorteil reduziert sich durch eine Umstellung auf bidirektionale Kanten, bleibt jedoch geringfügig erhalten, da *HiPE* in diesem Modell durch die hohe Zahl an verschiedenen Regeln bzw. Mustern und deren Abbildung auf jeweils eigene Threads den Vorteil seiner parallelisierten Ausführung besser ausnutzen kann.
Über den konkreten Speicherverbrauch lässt sich aufgrund der geringen Stichprobe an durchgeführten Simulationen sowie der starken Varianz des Speicherbedarfs keine qualitative Aussage treffen.

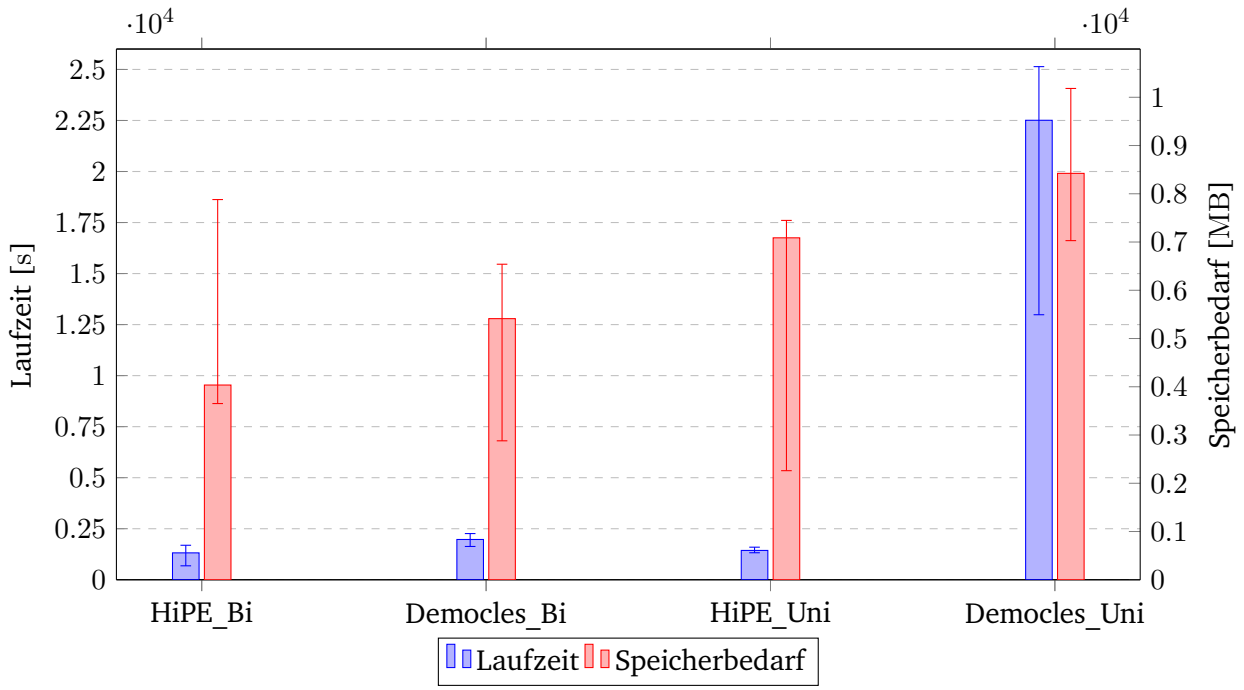
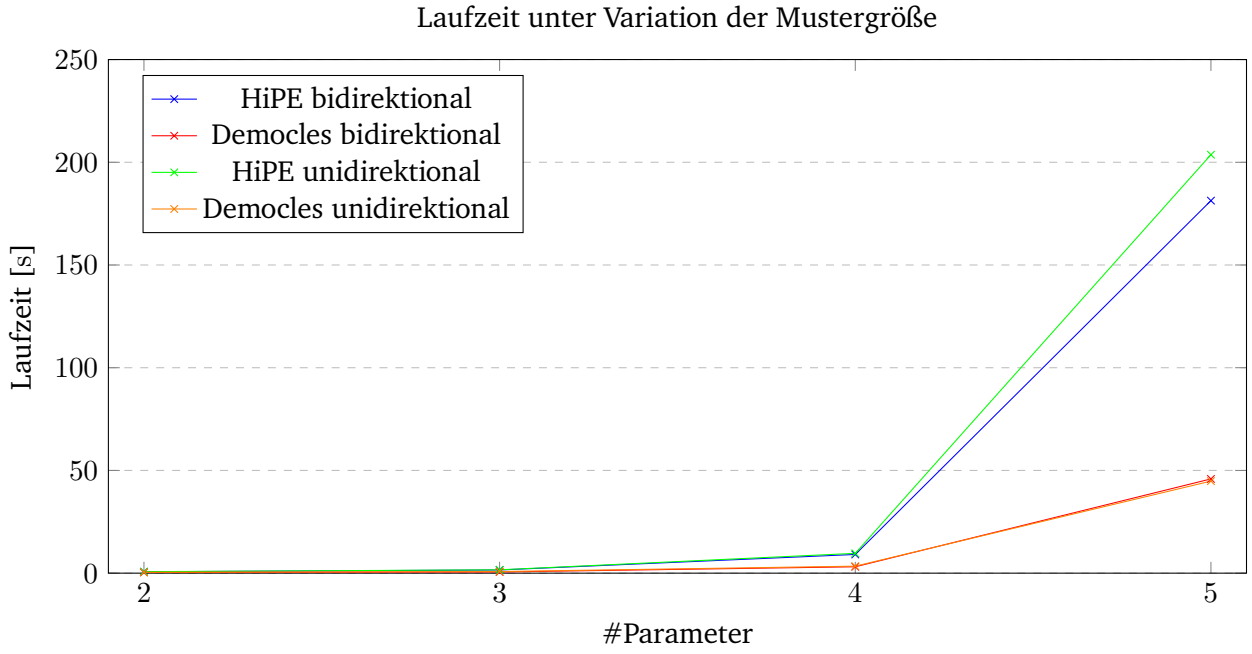


Abbildung 4.5.: Evaluationsergebnisse des Modells von *Proctor und Gray*

Variation der Mustergröße

Im folgenden wird der Einfluss von verschieden großen Mustern auf die Simulation betrachtet. Ergebnisse und Statistiken zu den Populationen im Modell nach Durchlauf der Simulation gibt es hier nicht, da keine Observer definiert wurden.
Die Mediane der benötigten Laufzeit und Höhe des Speicherverbrauchs in Abhängigkeit der Parameterzahl eines Musters zeigt Abbildung 4.6. Die Messreihen inklusive Fehlerabweichung sind aus Gründen der Übersichtlichkeit erneut in Abbildung 4.7 ausgelagert.
Es fällt sofort auf, dass die Zahl der Parameter nur bis zu einer Anzahl von fünf im Plot auftaucht. Dies ist einem beim Testgerät auftretenden *Heap Overflow* bei sechs Parametern in allen möglichen Konfigurationen zu schulden, der vermutlich durch die höhere Anzahl disjunkter Teilmuster der jeweiligen Modelle auftritt, da diese zu einem exponentiellen Anstieg der benötigten Ressourcen führen.
Bei den Laufzeitergebnissen wird sofort deutlich, dass die gewählte Kantenrepräsentation im Metamodell quasi keinen Einfluss auf die Laufzeit in Abhängigkeit der Parameterzahl hat. Die jeweiligen interpolierten Graphen der Simulationsdaten verhalten sich insbesondere unter Beachtung der Varianz in Abbildung 4.7a alle fast gleich. Ob das Metamodell auf uni- oder bidirektionalen Kanten basiert, macht bei diesem kleinen Modell keinen Unterschied, da alle Sites nur an genau eine andere Site binden können. Zudem schließt *Democles* hier bei weitem schneller ab als *HiPE*. Dies lässt sich darauf zurückführen, dass der Overhead durch Erstellung und Verwaltung von Threads zur Parallelisierung von *HiPE* in diesem kleinen Modell mit nur einer Regel den potenziell dadurch erreichbaren Performance-Vorteil überschattet.
Der Speicherverbrauch unterliegt erneut einer zu großen Varianz, um präzise Schlüsse aus den Messergebnissen ziehen zu können.



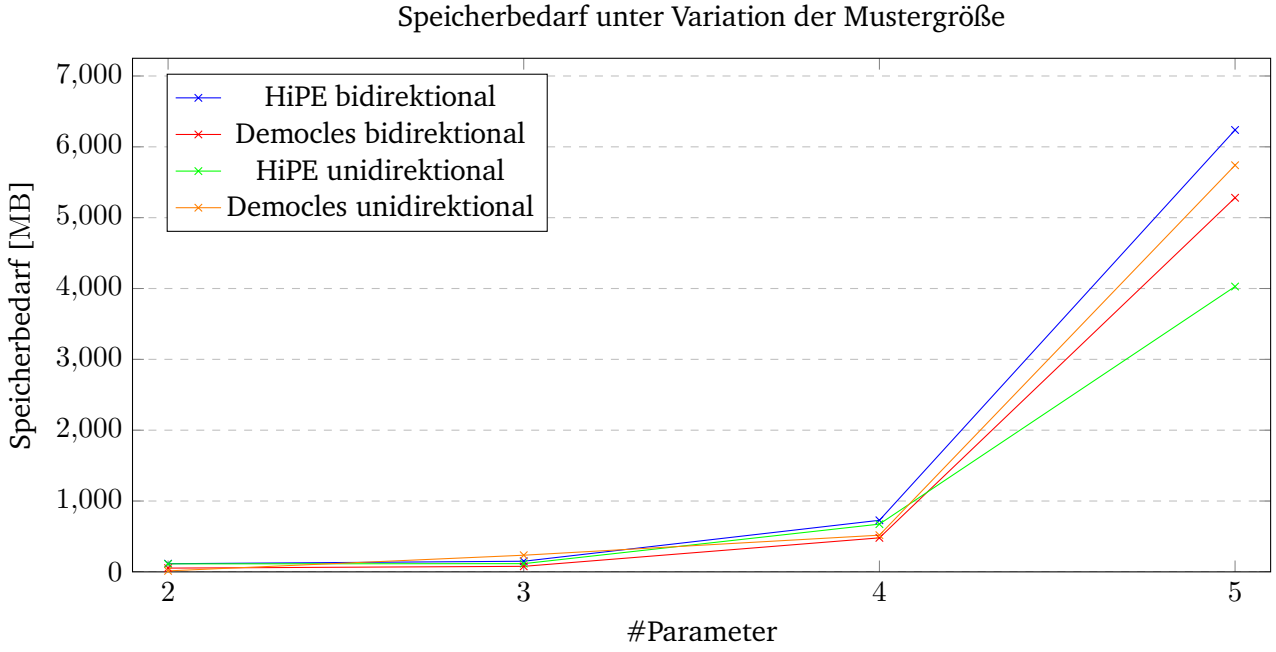
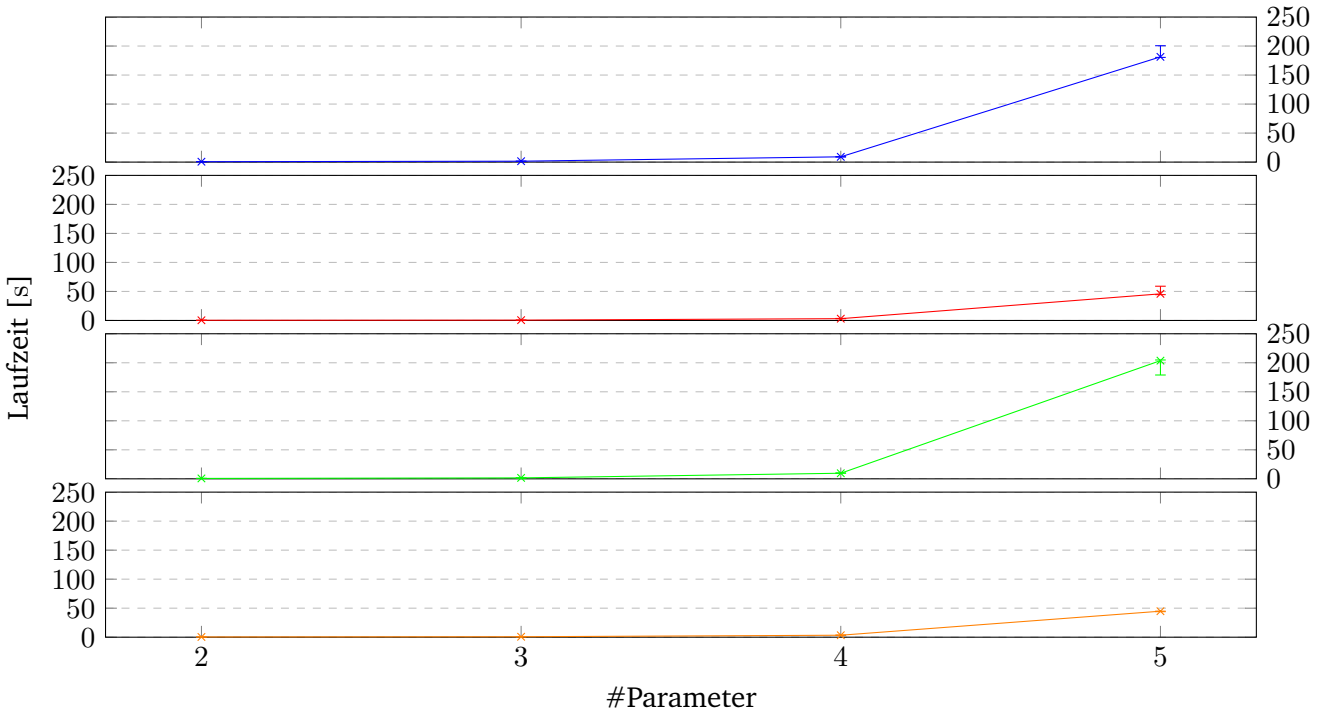
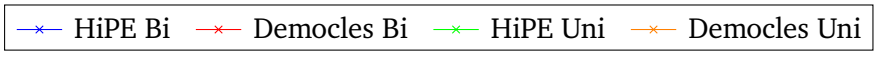
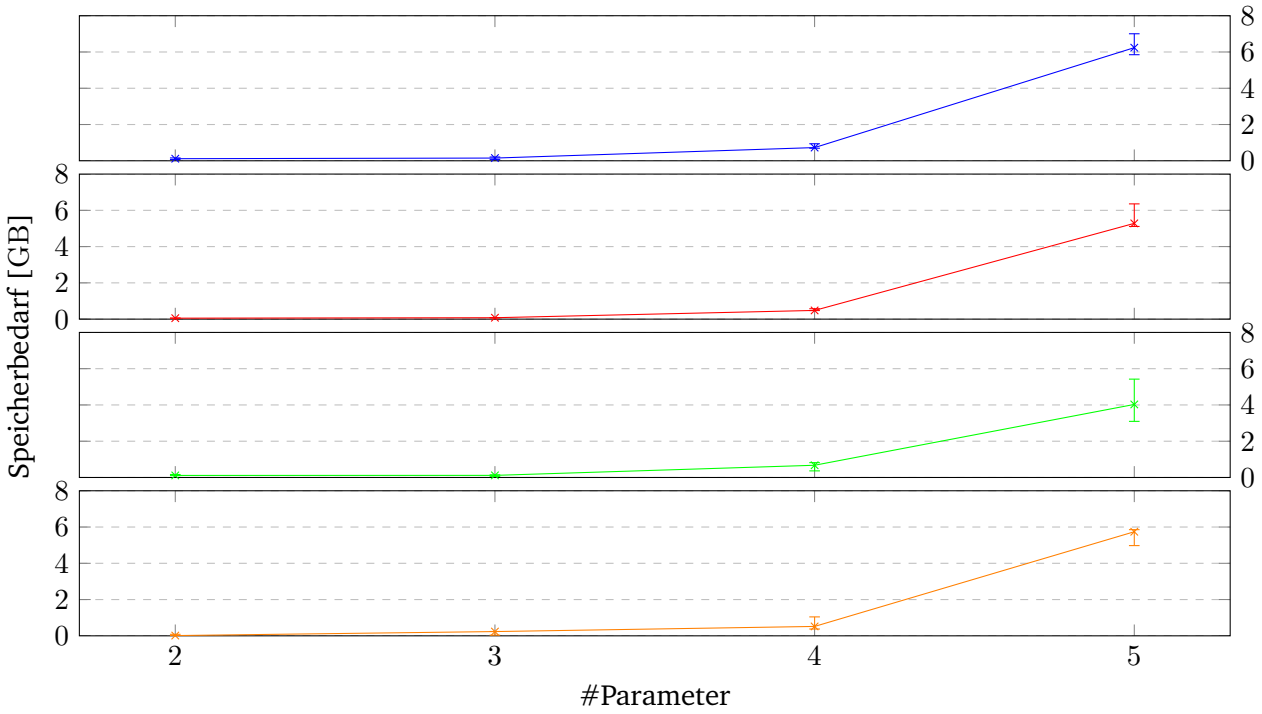


Abbildung 4.6.: Evaluationsergebnisse bei Variation der Patterngröße



(a) Laufzeit unter Variation der Mustergröße mit Fehlerbalken





(b) Speicherbedarf unter Variation der Mustergröße mit Fehlerbalken

Abbildung 4.7.: Evaluationsergebnisse bei Variation der Mustergröße

5. Verwandte Arbeiten

Im Biochemiekontext hat sich im Laufe der letzten Jahre ein beträchtliches Spektrum hilfreicher Tools zur differenzialgleichungs- oder regelbasierten Simulation biochemischer Prozesse gebildet. Einige solcher Werkzeuge wie *Kappa* oder *BioNetGen* wurden im Zuge dieser Arbeit bereits vorgestellt. Zur Einordnung des *Re.action*-Frameworks und der bereits vorgestellten sowie weiterer Tools in die Landschaft der biochemischen Simulationen und Modellierungen, wird der aktuelle *state of the art* hier nochmal kurz vorgestellt.

5.1. Kappa

Kappa[1] ist ein sehr moderner Vertreter der regelbasierten biochemischen Simulationen und sein grundlegendes κ-Kalkül diente als Basis für die Entwicklung von *Re.action*. Die Spezifikation ermöglicht die genaue Modellierung biochemischer Systeme und Prozesse mit umfangreichen Features wie zum Beispiel der Auflösung unterspezifizierter Verbindungen ohne Typangabe oder die Definition von Variablen, welche als Wert die zum betrachteten Zeitpunkt vorhandene Anzahl eines bestimmten Musters im System besitzt, wie es beides in *Re.action* nicht möglich ist.
Zur Spezifikation selbst gehört der entsprechende stochastische Simulator *KaSim*[1] und mittlerweile auch das statische Analyse-Tool *KaSa*. Zudem wird online ein übersichtliches und einfach zu bedienendes User-Interface zur Verfügung gestellt, mit dem Simulationen im Browser ausgeführt werden können.[2]
Die Spezifikation bietet in Hinblick auf Simulationsdirektiven weitere Features wie sogenannte *intervention directives*, die es ermöglichen, das System noch zur Simulationslaufzeit weiter zu manipulieren, indem man beispielsweise den Wert von Variablen ändert oder zusätzliche Agenten hinzufügt bzw. eliminiert. Diese Direktiven können zudem auch verschiedene Bedingungen besitzen, um als schleifenartige Anweisungen zu fungieren. Listing 5.1 zeigt einige dieser Befehle. Sie werden über das Schlüsselwort '`%mod`' eingeleitet. Der Befehl '`alarm`' nimmt eine reelle Zahl und ein auszuführendes

```
1 %var: 'n' |B(x[.])|
2 %mod: alarm 10.0 do $ADD 'n' C(x1{p}[.]); /* add some C */
3 %mod: |A()| > 1000 do $PLOTENTRY; repeat |B(x[_])| < 'n'
```

Listing 5.1: *Intervention directives* in *Kappa*

Event als Parameter. Die Zahl gibt das Zeitintervall an, nach dem die angegebene Aktion wiederholt werden soll. In diesem Beispiel werden dem System alle zehn Zeiteinheiten so viele Agenten C mit freier Site `x1` in Zustand `p` hinzugefügt, wie momentan Instanzen von Agenten B im System existieren,

[1] `github.com/Kappa-Dev/KaSim` [2] `kappalanguage.org/`

deren Site `x` ungebunden ist. In Zeile 3 wird eines der bereits erwähnten schleifenartigen Konstrukte verwendet. Hierzu wird eine Startbedingung und Schleifenbedingung definiert. Hier wird beispielsweise in jeder Iteration überprüft, ob die Anzahl an Agenten A im System die Grenze 1000 überschreitet. Ist dies der Fall, wird über den Befehl '`$PLOTENTRY`' in dieser und jeder weiteren Iteration der aktuelle Stand aller beobachteten Muster ausgegeben, bis die Schleifenbedingung verletzt wird. In diesem Fall also bis im System nicht mehr weniger Instanzen von Agenten B existieren, die eine gebundene Site `x` haben, als Instanzen mit einer freien Site `x`.

5.2. Systems Biology Markup Language (SBML)

Die *Systems Biology Markup Language* (SBML)[5] reiht sich ebenso in die prominenten Vertreter für Modellierungen in der Biochemiedomäne ein. Sie basiert auf der *eXtensible Markup Language* (XML) mit der allgemein hierarchisch strukturierte Daten dargestellt werden können.
Die SBML konkretisiert dieses Prinzip für die Modellierung biochemischer Prozesse. Hierin ist es möglich, verschiedene biochemische Spezies, Reaktionen, Variablen oder sogar physikalische Einheiten zu (re-)definieren. Bemerkenswert ist hierbei, dass auch über sogenannte '*compartments*' geometrische Informationen definiert werden können, da jedes *compartment* ein Volumen einer bestimmten Größe repräsentiert, deren Beziehungen untereinander auch spezifiziert werden können.
Obwohl auch diese Sprache textbasiert ist, bietet es sich nicht an, Modelle direkt wie in *Re.action* über sie zu spezifizieren, da das Format dafür viel zu verbos und die Sprache auch gar nicht dafür ausgelegt ist. Als Formatstandard für den generellen Austausch von Dateien für Modelle biochemischer Systeme, die zum Beispiel programmatisch erzeugt wurden und dennoch eine für Menschen lesbare Quelldatei besitzen sollen, ist *SBML* ein exzellentes Tool, weswegen es unter anderem als Standard für viele Datenbanken mit Modellen biochemischer Systeme dient.

5.3. BioNetGen (BNG)

Das *BioNetGen*-Framework[2] ist open-source und ebenfalls ein sehr prominenter Vertreter in der Welt der biochemischen Modellierungen und Simulationen. Die Syntax der zur Spezifikation von Regeln ist ähnliche zu *Kappa*, da Verbindungen ebenfalls indexbasiert modelliert werden.
Modelle der zugehörigen Sprache – die *BioNetGenLanguage* (BNGL) – sind durch festgelegte Abschnitte in '`begin ... end`'-Blöcken gut strukturiert. Generell ist das gesamte Framework ein sehr mächtiges Werkzeug, da es eine ganze Sammlung verschiedener open-source Simulationstools in sich vereint. Zum Beispiel integriert das Framework den stochastischen Simulator *NFSim*[41], wodurch weitere Simulationsoptionen geboten werden.
Aufgrund dieser großen Auswahl an Tools kann man im BNG-Framework zwischen verschiedenen Simulationsmethoden wählen, zum Beispiel ob die Simulation mittels eines Differentialgleichungssystems oder stochastisch bzw. regelbasiert durchgeführt werden soll. Dies macht den direkten Vergleich zwischen verschiedenen Ansätzen möglich.
Eine Vielzahl an Publikationen und weiterer Tools nutzt Modelle im *BioNetGen*-Format. Aus eben diesem Grund unterstützt auch das *Re.action*-Framework den Export von entsprechenden *Re.action*-Modellen in das *BNGL*-Format. *BioNetGen* selbst unterstützt den Export in viele andere geläufige Formate wie *MatLab*[42] oder die *Systems Biology Markup Language* (SBML) [5].

5.4. RuleBender

RuleBender[6] ist ein Tool zur Modellierung von regelbasierten Modellen auf Basis der *BioNetGenLanguage* und unterstützt dies mit einer grafischen Benutzeroberfläche und entsprechenden visuellen Repräsentationen zur Simulation und Analyse der betrachteten Modelle. So werden die verschiedenen Moleküle beispielsweise über sogenannte *contact maps* innerhalb der Benutzeroberfläche visualisiert, wie es Abbildung 5.1 zeigt. Dies erhöht die Nutzerfreundlichkeit der Eingabe von Mustern im Vergleich zu *Kappa* oder *Re.action* ungemein.

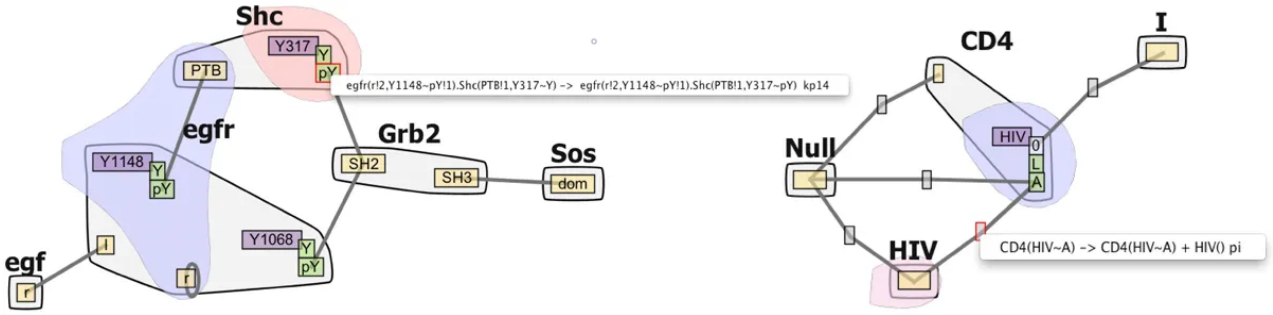


Abbildung 5.1.: Visualisierung von Molekülen in *RuleBender* via *contact maps* [6]

5.5. CellDesigner

CellDesigner[7] ist im Gegensatz zu den anderen bisher vorgestellten Tools kein textbasierter Editor zur Erstellung biochemischer Modelle, sondern bietet eine gänzlich grafische DSL zur Spezifikation. Diese Spezifikation beruht auf der *Systems Biology Graphical Notation* (SBGN) [43] und ermöglicht durch die visuelle Repräsentation intuitiv verständliche Modelle, welche leichter und schneller nachzuvollziehen sind als textuelle Repräsentationen. Damit diese Modelle durch Besitz zu vieler Informationen nicht zu unübersichtlich werden, sind sie in drei Module aufgeteilt. So wird wie Abbildung 5.2 nach molekularen Prozessen in Prozessdiagrammen, Beziehungen zwischen verschiedenen Entitäten in *Entity-Relationship-Modellen* und Verbindungen zwischen molekularen Aktivitäten unterschieden.
Außerdem dient *CellDesigner* durch den möglichen Zugriff auf verschiedene Biochemiedatenbanken wie *BioModels*[3] oder *pantherDB*[4] zugleich als Plattform für Modelle biochemischer Systeme. Für Modelle, die aus solchen Datenbanken abgerufen werden, unterstützt *CellDesigner* auch die *Systems Biology Markup Language* als Standard, da die meisten Datenbanken dieser Domäne darauf basieren. Die in *SBGN* formulierten Modelle werden zudem im *SBML*-Format gespeichert und verwaltet.

[3] `BioModels.net` [4] `pantherdb.org`

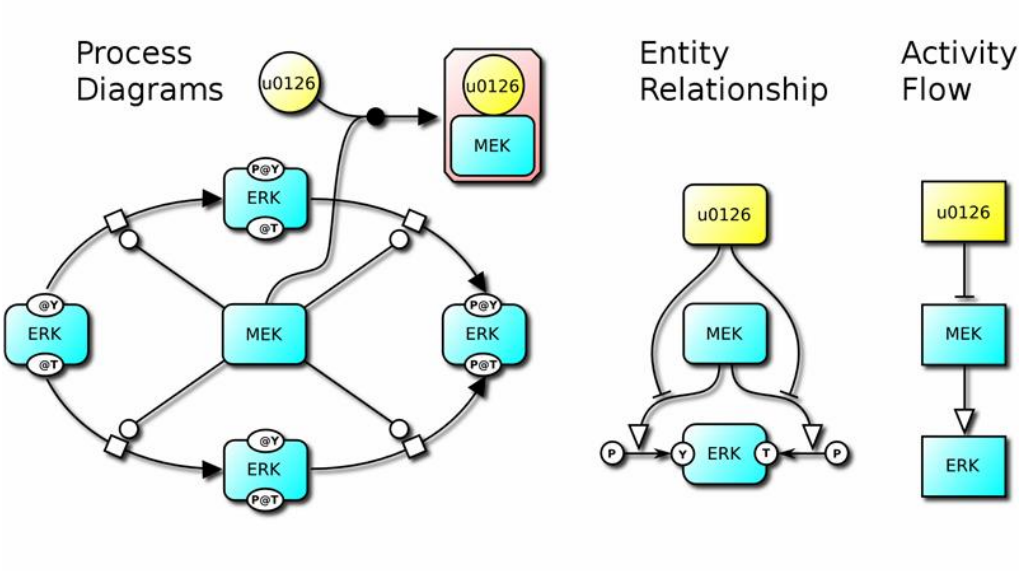


Abbildung 5.2.: *SBGNs* verschiedene grafische Repräsentationen biochemischer Modelle [43]

6. Zusammenfassung

Das primäre Ziel dieser Arbeit war die Entwicklung einer Spezifikation – konkret einer domänenspezifischen Sprache –, welche die in einem biochemischen System ablaufenden Prozesse unter Orientierung am κ-Kalkül vollständig als Regeln abbilden und modellieren kann. Das Sprachdesign sollte insbesondere in Hinblick auf seine allgemeine Usability, also seine intuitive Verständlichkeit und möglichst kompakte Formulierung von Ausdrücken, optimiert werden. Die zentrale Herausforderung war es hierbei, einen möglichst sinnvollen Kompromiss zwischen den genannten Parametern zu finden. Dieser Prozess kulminierte in der neuen Sprache *Re.action*.
Um einen praktischen Nutzen aus dieser Sprache zu ziehen, wurde ein Framework entwickelt, welches die Einspeisung des spezifizierten Modells in das Simulationstool *SimSG*[4] ermöglicht, um die modellierten regelbasierten Simulationen auch ausführen zu können. Hierzu mussten Modelltransformationen des Sprachmodells vorgenommen werden, die am Ende ein zu *SimSG* konformes Simulationsmodell liefern. Dieses besteht aus verschiedenen Teilmodellen, in denen die zuvor definierten biochemischen Reaktionsregeln in (Sub-)Graphrepräsentationen und die zugehörigen Transformationsregeln umgewandelt wurden.
Abschließend wurde die entwickelte Sprache bezüglich der für die Entwicklung richtungsweisenden Parameter evaluiert und mit anderen etablierten Sprachen der Biochemiedomäne verglichen. Zudem wurden die Pattern-Matching-Tools *Democles* und *HiPE* des CASE-Tools *eMoflon*[3], welches in *SimSG* zur Realisierung der Simulation integriert ist, mithilfe der neuen Spezifikation in Bezug auf Laufzeit und Speicherverbrauch bei Anwendung auf verschiedene Modelle gegeneinander evaluiert.

Die Entwicklung der Spezifikation *Re.action* sollte hierbei auch insbesondere für Nutzer intuitiv verständlich sein, die kein Vorwissen zu Programmiersprachen oder Software-Entwicklung besitzen, da der Anwendungskontext dieses Frameworks hauptsächlich in der Biochemie liegt. Hierzu sollten primär Elemente der natürlichen, menschlichen Sprache als Leitfaden für die Grammatik dienen. Da dies an vielen Stellen jedoch mit dem Ziel einer möglichst kompakten Syntax kollidiert, wurde diese Vorgehensweise vielerorts darauf reduziert, lediglich einzelne Operatoren zu nutzen, die dafür allerdings eine hohe assoziative Semantik aufweisen. Die Anforderungen an die Spezifikation wurden dabei vordergründig durch das von *Danos et al.*[8] formulierte κ-Kalkül bestimmt. Dieses legt die verschiedenen Komponenten der Spezifikation sowie eine Klassifikation verschiedener Bindungszustände fest. Auf dieser Basis wurden eine Modellsignatur zur Definition von Agententypen samt Sites und derer internen Zustände, Initialisierungen als Anfangsbedingungen, Variablen, Regeln mit eigener Reaktionssignatur und den entsprechenden als Muster formulierten Vor- und Nachbedingungen, Observer zur statistischen Erfassung von Simulationsdaten, Terminate-Direktiven als Abbruchbedingungen der Simulation sowie Komplexe als gruppierendes und kompaktifizierendes Element für Regeln eingeführt. Maßgeblich bei der Definition von Mustern war hierbei in Anlehnung an das κ-Kalkül das sogenannte „*don't care, don't write*"-Prinzip, nach dem nur die Agenten und Sites eines Musters formuliert werden müssen, die auch relevante Informationen zum Muster beitragen. Ist z.B der interne Zustand einer Site beliebig, so wird er bei Eingabe des Musters gar nicht erst aufgeführt. Die Formulierung des gesamten

Modells erfolgte hierbei mittels des Tools *Xtext* als Plug-in für die Entwicklungsumgebung *Eclipse*, die somit direkt als Editor samt Syntax-Highlighting und Support für Warnungen und Fehlermeldungen während der Spracheingabe dient.
Im Vergleich zu anderen Sprachen aus der Domäne regelbasierter biochemischer Simulationen wie *Kappa*[1] und der *BioNetGenLanguage (BNGL)*[2] konnte mit *Re.action* eine Alternative geschaffen werden, die im Gegensatz zu den beiden genannten Sprachen keine Darstellung von Mustern über eine Indexkennzeichnung der Sites von beteiligten Agenten verfolgt, sondern Muster primär durch die Summe paarweiser Verbindungen der Sites beteiligter Agenten repräsentiert.
Die Herausforderung der kompakten Spezifikation wird von *Kappa* und *BNGL* in manchen Punkten besser, an anderen Stellen wiederum schlechter gelöst. Ob die intuitive Verständlichkeit von *Re.action* speziell für programmierunerfahrene Anwender aus der Biochemie höher ist, hängt stark von den subjektiven Präferenzen der Anwender ab und wird sich im Laufe der Zeit zeigen.
Die Transformation des originalen Sprachmodells in ein zu *SimSG* konformes Simulationsmodell, welches auf graphbasierten Modellen aufbaut, wurde in zwei Teiltransformationen aufgespalten, indem vor der finalen Transformation zuerst ein Zwischenmodell generiert wird. Durch diese Teilung in zwei voneinander unabhängige Arbeitsschritte erhöht sich die Wartbarkeit und Erweiterbarkeit des gesamten Frameworks enorm.
Die erste Subtransformation entfernt hierbei Redundanzen, welche durch die Generierung des Sprachmodells auf Grundlage der Sprachgrammatik entstehen, und vollzieht erste Schritte zur Vorbereitung der abschließenden Transformation. Dies geschieht beispielsweise, indem die in *Re.action* u.a. auf Basis mehrerer Bindungen formulierten Muster in eine Menge aus Agenteninstanzen übersetzt werden, welche für das endgültige Simulationsmodell anschließend nur noch in Knoten des resultierenden Graphen umgewandelt werden müssen. Durch das hieraus resultierende Zwischenmodell wurde zusätzlich die Unterstützung einer Übersetzung von *Re.action* in die im Kontext biochemischer Simulationen omnipräsente *BioNetGenLanguage (BNGL)* möglich, wodurch *Re.action*-Modelle sehr leicht in zahlreiche andere Simulationsumgebungen integriert werden können. So können Anwender trotzdem weiterhin alle *BNGL*-unterstützenden Tools nutzen, aber nach Bedarf wählen, ob sie zur Spezifikation *BNGL* oder *Re.action* verwenden möchten.
Die zweite Teiltransformation vollzieht die Übersetzung von diesem Zwischenmodell zum ausführbaren Simulationsmodell, das sowohl die Anfangs- und Abbruchbedingungen für die Simulation selbst, als auch eine Menge an Graphtransformationsregeln enthält, welche die biochemischen Reaktionsregeln als Graphenmuster repräsentieren. Dort werden ihnen zudem weitere Attribute wie zum Beispiel eine Reaktionsrate zugeordnet.
Das somit vollständige Framework ermöglicht Simulationen in *SimSG* mit den beiden Pattern Matchern *Democles* und *HiPE*. Unter Einsatz des neuen Frameworks konnten effizient verschiedene Modelle zur Evaluation dieser beiden Engines erstellt werden.
Im Zuge jener Evaluation konnten die Auswirkungen der Umstrukturierung des grundlegenden Metamodells für die Simulationsmodelle von unidirektionalen zu bidirektionalen Kanten untersucht und ein konkreter Vergleich zwischen den beiden genutzten Pattern Matching Engines durchgeführt werden. Dies führte zum einen zu der Erkenntnis, dass bei bidirektionalen Kanten die kleineren *IBeX*-Patterns und starke Typisierung der Kanten trotz der größeren Anzahl an *Negative Application Conditions* in den zugehörigen Patterns zu einer Verkürzung der Laufzeit führen. Diese wird insbesondere bei großen Modellen wie dem von *Proctor und Gray*[9] unter Nutzung der *Democles*-Engine deutlich.
Zum anderen zeigte sich *HiPE* durch seine parallele Ausführungsweise bei großen Modellen als die etwas kostengünstigere Alternative. Bei der Ausführung von *HiPE* mit uni- und bidirektionalen Kanten zeigten sich keine signifikanten Unterschiede.

6.1. Future Work

Da die Spezifikation samt Framework für *SimSG* entwickelt wurde, das auf *eMoflon* aufbaut und die dort integrierten *general-purpose Pattern Matcher* nutzt, statt hochspezialisierte Werkzeuge wie zum Beispiel *Kappa*, verbirgt sich hier noch ungenutztes Potenzial. Somit könnte die Sprache ohne sonderlichen Aufwand erweitert werden, um Bedingungen für komplexere Regelanwendungsstrategien wie beispielsweise bestimmte Werte für Attribute der verschiedenen Knoten respektive Agenten zu unterstützen.
Ebenso ist die Implementierung des Frameworks unter anderem durch seinen zweistufigen Transformationsprozess stark skalierbar. Das Zwischenmodell enthält die Informationen aus der Spezifikation schon in kompakter Form und Darstellung, was den Aufwand für Transformationen in weitere Simulationsmodelle bzw. Spezifikationen erleichtert. Diese Eigenschaft kann dazu genutzt werden, die hier durchgeführte Evaluation von Pattern Matchern auf weitere Engines auszuweiten, wie es zum Beispiel schon durch den möglichen Export in das *BNGL*-Format möglich ist. In diesem Zuge kann auch direkt der Stichprobenumfang an durchgeführten Messungen bzw. Simulationen erhöht werden, sodass präzisere Aussagen als in der kleinen Evaluation dieser Arbeit getroffen werden können. Ebenso wäre es von Vorteil, eine umfassend angelegte Nutzerstudie zur Evaluation der *Re.action*-Sprache durchzuführen, um deren Syntax und Funktionen in Bezug auf die Bedürfnisse von Anwendern aus dem Biochemiekontext zu erweitern und anzupassen.
Des Weiteren könnte das Framework mitsamt *SimSG* erweitert werden, um zusätzliche Direktiven zu ermöglichen, welche den genaueren Simulationsablauf manipulieren und steuern können. Dazu zählt beispielsweise das Hinzufügen oder Eliminieren von Agenten zur Simulationslaufzeit zu bestimmten Zeitpunkten oder nach Erreichen bestimmter Bedingungen, wie es in *Kappa* durch die dortigen *intervention directives* bereits möglich ist.
Zudem könnte man auch die Art der Variablen von numerischen Werten bzw. arithmetischen Ausdrücken auf Mustervariablen ausweiten, die anstelle von Vor- oder Nachbedingungen in Regeln erscheinen können oder in arithmetischen Ausdrücken die zum Zeitpunkt der Auswertung des Ausdrucks im Modell gefundenen Matches als numerischen Wert repräsentieren.
Ebenso kann die in *Eclipse* zur Verfügung gestellte grafische Benutzeroberfläche, welche auch den Texteditor zur Eingabe der Sprache selbst enthält, um weitere Module erweitert werden. Vorstellbar wäre hier ein Modul zur Visualisierung verschiedener Agenten- bzw. Molekülbeziehungen bei Auswahl eines bestimmten Musters im Texteditor, wie es schon *RuleBender* vorführt.

A. Appendix

A. Backus-Naur-Form der *Re.action*-Grammatik

⟨*Model*⟩ ::= ⟨*Components*⟩

⟨*Components*⟩ ::= ⟨*Component*⟩ | ⟨*Component*⟩ ⟨*Components*⟩

⟨*Component*⟩ ::= ⟨*AgentDeclaration*⟩
| ⟨*Rule*⟩
| ⟨*Complex*⟩
| ⟨*Initialisation*⟩
| ⟨*Observer*⟩
| ⟨*Variable*⟩
| ⟨*Command*⟩

⟨*AgentDeclaration*⟩ ::= '`agent`' ⟨*Agents*⟩

⟨*Agents*⟩ ::= ⟨*Agent*⟩ | ⟨*Agent*⟩ '`,`' ⟨*Agents*⟩

⟨*Agent*⟩ ::= ⟨*AgentName*⟩ | ⟨*AgentName*⟩ '`:`' ⟨*Sites*⟩

⟨*Sites*⟩ ::= ⟨*Site*⟩ | ⟨*Sites*⟩ '`,`' ⟨*Site*⟩

⟨*Site*⟩ ::= ⟨*SiteName*⟩ | ⟨*SiteName*⟩ '`(`' ⟨*States*⟩ '`)`'

⟨*States*⟩ ::= ⟨*SiteState*⟩ | ⟨*SiteState*⟩ '`,`' ⟨*States*⟩

⟨*SiteState*⟩ ::= ⟨*StateName*⟩

⟨*Rule*⟩ ::= '`rule`' ⟨*RuleHead*⟩ ⟨*RuleBody*⟩

⟨*RuleHead*⟩ ::= ⟨*RuleName*⟩ ⟨*Signature*⟩ '`:`'

⟨*Signature*⟩ ::= '`(`' ⟨*AgentInstanceDecls*⟩ '`)`'

⟨*AgentInstanceDecls*⟩ ::= ⟨*AgentInstanceDecl*⟩ | ⟨*AgentInstanceDecl*⟩ '`,`' ⟨*AgentInstanceDecls*⟩

⟨*AgentInstanceDecl*⟩ ::= ⟨*AgentInstanceName*⟩ '`:`' ⟨*AgentName*⟩

⟨*RuleBody*⟩ ::= ⟨*Pattern*⟩ ⟨*RuleType*⟩ ⟨*Pattern*⟩ '`@`' ⟨*Rates*⟩

⟨*Rates*⟩ ::= ⟨*Number*⟩ | ⟨*Number*⟩ '`,`' ⟨*Number*⟩

⟨*Pattern*⟩ ::= '`_`' | ⟨*PatternElements*⟩

⟨*PatternElements*⟩ ::= ⟨*PatternElement*⟩ | ⟨*PatternElement*⟩ '`,`' ⟨*PatternElements*⟩

⟨*PatternElement*⟩ ::= ⟨*AgentInstanceName*⟩
| ⟨*Bond*⟩

⟨*Bond*⟩ ::= ⟨*LeftBondSide*⟩ ⟨*BondType*⟩ ⟨*RightBondSide*⟩
| ⟨*LeftBondSide*⟩ ⟨*Wildcard*⟩

⟨*BondType*⟩ ::= '`//`' | '`+`'

⟨*Wildcard*⟩ ::= '`+?`' | '`//0`'

⟨*LeftBondSide*⟩ ::= ⟨*AgentInstanceName*⟩ '`.`' ⟨*SiteInstance*⟩

⟨*RightBondSide*⟩ ::= ⟨*AgentInstanceName*⟩ '`.`' ⟨*SiteInstance*⟩
| ⟨*AgentName*⟩ '`.`' ⟨*SiteInstance*⟩

⟨*SiteInstance*⟩ ::= ⟨*SiteName*⟩
| ⟨*SiteName*⟩ '`(`' ⟨*StateName*⟩ '`)`'

⟨*RuleType*⟩ ::= '`=>`' | '`<=>`'

⟨*Complex*⟩ ::= '`complex`' ⟨*Signature*⟩ '`{`' ⟨*ComplexElements*⟩ '`}`'

⟨*ComplexElements*⟩ ::= ⟨*ComplexElement*⟩ | ⟨*ComplexElement*⟩ ⟨*ComplexElements*⟩

⟨*ComplexElement*⟩ ::= ⟨*Complex*⟩
| ⟨*ComplexRule*⟩
| ⟨*Variable*⟩
| ⟨*Observer*⟩
| ⟨*ComplexObserver*⟩

⟨*ComplexRule*⟩ ::= ⟨*Rule*⟩
| '`rule`' ⟨*ComplexRuleHead*⟩ ⟨*RuleBody*⟩

⟨*ComplexRuleHead*⟩ ::= ⟨*RuleName*⟩ '`:`'

⟨*Variable*⟩ ::= '`var`' ⟨*VariableName*⟩ '`=`' ⟨*Number*⟩

⟨*Initialisation*⟩ ::= ⟨*InitialisationHead*⟩ ⟨*InitialisationBody*⟩

⟨*InitialisationHead*⟩ ::= '`init`' ⟨*Number*⟩

⟨*InitialisationBody*⟩ ::= ⟨*Signature*⟩ '`:`' ⟨*Pattern*⟩

⟨*Observer*⟩ ::= '`observe`' ⟨*ObserverName*⟩ ⟨*Signature*⟩ '`:`' ⟨*Pattern*⟩

⟨*ComplexObserver*⟩ ::= observe ⟨*ObserverName*⟩ '`:`' ⟨*Pattern*⟩

⟨*Command*⟩ ::= terminate ⟨*TerminateBody*⟩

⟨*TerminateBody*⟩ ::= ⟨*TerminateTime*⟩
| ⟨*TerminateIterations*⟩
| ⟨*TerminateCount*⟩

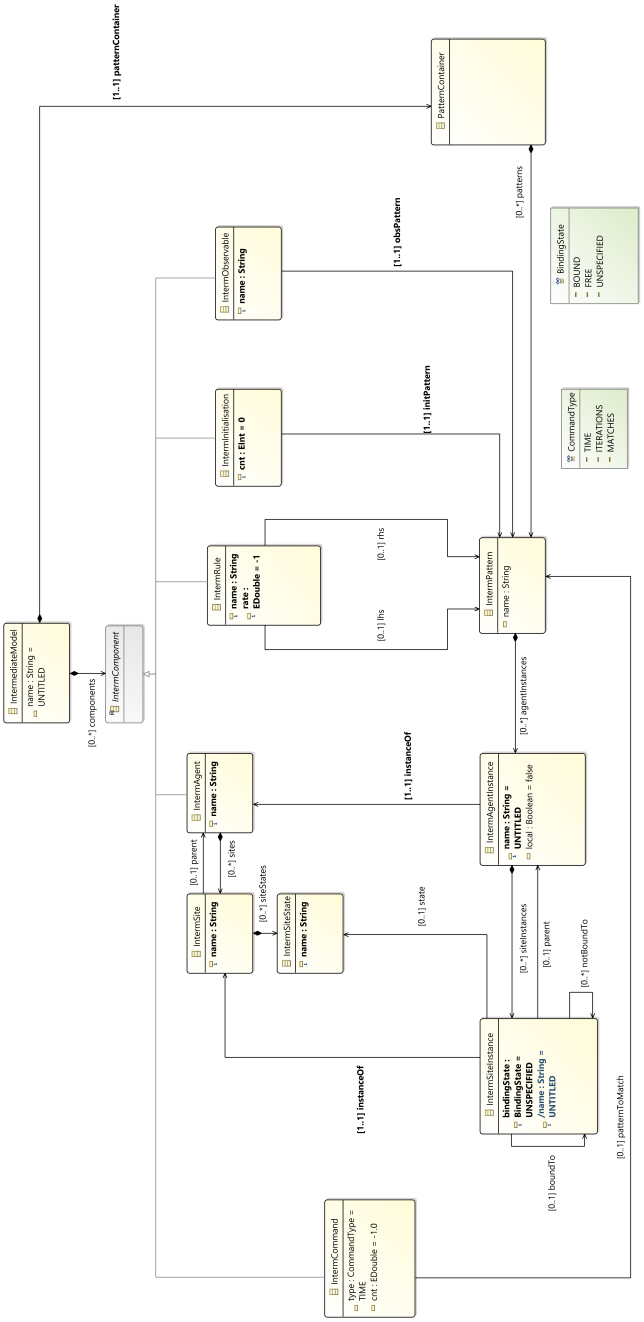
⟨*TerminateTime*⟩ ::= '`time=`' ⟨*Number*⟩

⟨*TerminateIterations*⟩ ::= '`iterations=`' ⟨*Number*⟩

⟨*TerminateCount*⟩ ::= ⟨*Signature*⟩ '`:`' ⟨*Pattern*⟩ '`matches=`' ⟨*Number*⟩

Anstelle von <*Number*> können in der tatsächlichen Spezifikation beliebige arithmetische Ausdrücke mit Variablen – wie in Kapitel 3.1 beschrieben – stehen. Diese sind hier der Einfachheit halber ausgelassen.
Alle in der BNF auftauchenden Namen sind hierbei beliebige *Identifier*, die mit einem Buchstaben oder Underscore beginnen und auch Ziffern enthalten können.

B. Klassendiagramm des Zwischenmodells



C. Goldbeter-Koshland-Loop in *Re.action*

```
agent K: a
agent P: a
agent T: x(u,p), y(u,p)

var  initAmount = 100 # Wird skaliert zu 200 , 400 , 800 und 1600

init initAmount (k: K, p: P, t: T):     k, p, t

complex (k: K, t: T){
    rule KT_x: k.a//0 , t.x//0  <=> k.a+t.x          @1.0, 10.0
    rule Tp_x: k.a+t.x(u)        => k.a+t.x(p)       @1.0
    rule KT_y: k.a//0 , t.y//0  <=> k.a+t.y          @1.0, 10.0
    rule Tp_y: k.a+t.y(u)        => k.a+t.y(p)       @1.0
}

complex (p: P, t:T){
    rule PT_x: p.a//0 , t.x//0  <=> p.a+t.x          @1.0, 10.0
    rule Tu_x: p.a+t.x(p)        => p.a+t.x(u)       @1.0
    rule PT_y: p.a//0 , t.y//0  <=> p.a+t.y          @1.0, 10.0
    rule Tu_y: p.a+t.y(p)        => p.a+t.y(u)       @1.0
}

observe T_pp (t: T):            t.x(p), t.y(p)
observe T_pp_unbound (t: T):    t.x(p)//0 , t.y(p)//0

terminate iterations = 100
```

D. GSK3b / Mdm2 Modell in *Re.action*

Die Anmerkungen in den Kommentaren der verschiedenen Regelkomplexe referenzieren die Nummern der Regeln, wie sie im Regelset der Arbeit von *Proctor und Gray* [9] gelistet wurden.

```
agent GSK3b: x
agent P53: b(u, p, ub1, ub2, ub3, ub4); P53mRNA; P53DUB
agent Tau: phos(u, p1, p2); AggTau: y
agent MT: l
agent PPT
agent NFT
agent Proteasome: x
agent Mdm2: z(u, p), ubi(unb, ub1, ub2, ub3, ub4); Mdm2mRNA; Mdm2DUB
agent Abeta; AggAbeta: y
agent AbetaPlaque
agent ROS; BasalROS
agent E1: x(unb, ubi); E2: x(unb, ubi)
agent Ub
agent AMP; ADP; ATP
agent ATMA; ATMI
agent DamDNA
agent IR

#Initialisation Vars
var gsk3bInit        =   100
var p53Init          =   1
var mdm2Init         =   1
var mdm2_p53Init     =   19
var mdm2_mRNAInit    =   2
var p53_mRNAInit     =   2
var UbInit           =   800
var E1Init           =   2
var E2Init           =   2
var p53DubInit       =   40
var mdm2DubInit      =   40
var proteasomeInit   =   100
var atmiInit         =   40
var basalRosInit     =   2
var mtTauInit        =   20
var pptInit          =   10
var atpInit          =   2000
var adpInit          =   200
var ampInit          =   200

#Constants
var ksynp53mRNA                =   0.001
var kdegp53mRNA                =   1.00E-04
var ksynMdm2mRNA               =   5.00E-04
var kdegMdm2mRNA               =   5.00E-04
var ksynMdm2mRNAGSK3bp53       =   7.00E-04
var ksynp53                    =   0.007
var kdegp53                    =   0.005
var kbinMdm2p53                =   0.001155
var krelMdm2p53                =   1.16E-02
var kbinGSK3bp53               =   2.00E-06
var krelGSK3bp53               =   0.002
var ksynMdm2                   =   4.95E-04
var kdegMdm2                   =   0.01
var kbinE1Ub                   =   2.00E-04
var kbinE2Ub                   =   0.001
var kp53Ub                     =   5.00E-05
var kp53PolyUb                 =   0.01
var kbinProt                   =   2.00E-06
var kactDUBp53                 =   1.00E-07
var kactDUBProtp53             =   1.00E-04
```

```
61  var kactDUBMdm2              =   1.00E-07
62  var kMdm2Ub                  =   4.56E-06
63  var kMdm2PUb                 =   6.84E-06
64  var kMdm2PolyUb              =   0.00456
65  var kdam                     =   0.08
66  var krepair                  =   2.00E-05
67  var kactATM                  =   1.00E-04
68  var kinactATM                =   5.00E-04
69  var kphosp53                 =   2.00E-04
70  var kdephosp53               =   0.5
71  var kphosMdm2                =   2
72  var kdephosMdm2              =   0.5
73  var kphosMdm2GSK3b           =   0.005
74  var kphosMdm2GSK3bp53        =   0.5
75  var kphospTauGSK3bp53        =   0.1
76  var kphospTauGSK3b           =   2.00E-04
77  var kdephospTau              =   0.01
78  var kbinMTTau                =   0.1
79  var krelMTTau                =   1.00E-04
80  var ksynTau                  =   8.00E-05
81  var kdegTau                  =   1.00E-02
82  var kbinTauProt              =   1.93E-04
83  var kdegTau20SProt           =   0.01
84  var kaggTau                  =   1.00E-08
85  var kaggTauP1                =   1.00E-08
86  var kaggTauP2                =   1.00E-07
87  var ktangfor                 =   0.001
88  var kprodAbeta               =   5.00E-05
89  var kinhibprot               =   1.00E-05
90  var kdegAbeta                =   1.00E-04
91  var kaggAbeta                =   1.00E-08
92  var kpf                      =   0.001
93  var ksynp53mRNAAbeta         =   1.00E-05
94  var kdamROS                  =   1.00E-05
95  var kdamBasalROS             =   1.00E-09
96  var kgenROSAbeta             =   1.00E-05
97  var kproteff                 =   1
98
99  #Initialisations
100 init gsk3bInit (gsk: GSK3b):                gsk//0
101 init p53Init (p53: P53):                    p53//0
102 init mdm2Init (mdm2: Mdm2):                 mdm2//0
103 init mdm2_p53Init (mdm2: Mdm2, p53: P53):   p53.b(u)+mdm2.z(u)
104 init mdm2_mRNAInit (mdm2mRna: Mdm2mRNA):    mdm2mRna//0
105 init p53_mRNAInit (p53mRna: P53mRNA):       p53mRna
106 init UbInit (ub: Ub):                       ub
107 init E1Init (e1: E1):                       e1//0
108 init E2Init (e2: E2):                       e2//0
109 init p53DubInit (p53Dub: P53DUB):           p53Dub
110 init mdm2DubInit (mdm2Dub: Mdm2DUB):        mdm2Dub
111 init proteasomeInit (prot: Proteasome):     prot//0
112 init atmiInit (atmi: ATMI):                 atmi
113 init basalRosInit (basalRos: BasalROS):     basalRos
114 init mtTauInit (tau: Tau, mt: MT):          tau.phos+mt.l
115 init pptInit (ppt: PPT):                    ppt//0
116 init atpInit (atp: ATP):                    atp
117 init adpInit (adp: ADP):                    adp
118 init ampInit (amp: AMP):                    amp
119
120 #rules 1-3
121 complex(gsk: GSK3b, p53: P53, mdm2mRNA: Mdm2mRNA){
122     rule GSK3b_p53U: gsk.x//0, p53.b(u)//0 <=> gsk.x+p53.b(u)   @kbinGSK3bp53, krelGSK3bp53
123     rule GSK3b_p53P: gsk.x//0, p53.b(p)//0 <=> gsk.x+p53.b(p)   @kbinGSK3bp53, krelGSK3bp53
124     rule Mdm2mRNASynthU: gsk.x+p53.b(u) => gsk.x+p53.b(u), mdm2mRNA//0  @ksynMdm2mRNAGSK3bp53
125     rule Mdm2mRNASynthP: gsk.x+p53.b(p) => gsk.x+p53.b(p), mdm2mRNA//0  @ksynMdm2mRNAGSK3bp53
126 }
127
```

```
#mdm2gsk3phosphorylation ____ rules 4–5
complex(gsk: GSK3b, p53_1: P53, p53_2:P53, mdm2: Mdm2){
    rule Mdm2GSK3Phospho1:  mdm2.z(u)+p53_1.b(ub4), mdm2.ubi(unb)//0, gsk//0
        => mdm2.ubi(unb)//0, mdm2.z(p)+p53_1.b(ub4), gsk//0          @kphosMdm2GSK3b

    rule Mdm2GSK3Phospho2_3:mdm2.z(u)+p53_1.b(ub4), mdm2.ubi(unb)//0, gsk.x+p53_2.b
        => mdm2.z(p)+p53_1.b(ub4), mdm2.ubi(unb)//0, gsk.x+p53_2.b  @kphosMdm2GSK3bp53
}

#rules 6–10
complex(tau: Tau, prot: Proteasome){
    rule TauSynthesis:   _                              => tau//0       @ksynTau
    rule Proteasome_Tau:
        tau.phos(u)//0, prot.x//0 => tau.phos(u)+prot.x                 @kbinTauProt

    rule TauDegradation: prot.x+tau.phos(u)             => prot.x//0    @kdegTau
    rule TauMT(mt: MT):
        tau.phos(u)//0, mt.l//0     <=> tau.phos(u)+mt.l                @kbinMTTau,krelMTTau
}

#tau (de–)phosphorylation ____ rules 11–13
complex(tau: Tau, p53: P53, gsk: GSK3b){
    rule TauPhosphorylation1:   gsk.x+p53.b(u), tau.phos(u)//0
        => gsk.x+p53.b(u), tau.phos(p1)//0                              @kphospTauGSK3bp53

    rule TauPhosphorylation3:   gsk.x+p53.b(p), tau.phos(u)//0
        => gsk.x+p53.b(p), tau.phos(p1)//0                              @kphospTauGSK3bp53

    rule TauPhosphorylation2:   gsk.x+p53.b(u), tau.phos(p1)//0
        => gsk.x+p53.b(u), tau.phos(p2)//0                              @kphospTauGSK3bp53

    rule TauPhosphorylation4:   gsk.x+p53.b(p), tau.phos(p1)//0
        => gsk.x+p53.b(p), tau.phos(p2)//0                              @kphospTauGSK3bp53

    rule TauPhosphorylation5:   gsk//0, tau.phos(u)//0
        => gsk//0, tau.phos(p1)//0                                      @kphospTauGSK3b

    rule TauPhosphorylation6:   gsk//0, tau.phos(p1)//0
        => gsk//0, tau.phos(p2)//0                                      @kphospTauGSK3b
}

rule TauDephosphorylation(tau: Tau, ppt: PPT):
    tau.phos(p2)//0, ppt => tau.phos(p1)//0, ppt                        @kdephospTau

#tau aggregation ____ rules 14–15
complex(tau1: Tau, tau2: Tau, agg1: AggTau, agg2: AggTau){
    rule TauAgg1:
        tau1.phos(u)//0,  tau2.phos(u)//0   => agg1//0, agg2//0     @kaggTau/2

    rule TauAgg2:
        tau1.phos(u)//0,  agg1//0           => agg1//0, agg2//0     @kaggTau

    rule TauP1Agg1:
        tau1.phos(p1)//0, tau2.phos(p1)//0  => agg1//0, agg2//0     @kaggTauP1/2

    rule TauP1Agg2:
        tau1.phos(p1)//0, agg1//0           => agg1//0, agg2//0     @kaggTauP1

    rule TauP2Agg1:
        tau1.phos(p2)//0, tau2.phos(p2)//0  => agg1//0, agg2//0     @kaggTauP2/2

    rule TauP2Agg2:
        tau1.phos(p2)//0, agg1//0           => agg1//0, agg2//0     @kaggTauP2
}
```

```
195 #tangle formations ____ rule 16
196 complex(agg1: AggTau, agg2: AggTau, nft1: NFT, nft2: NFT){
197     rule tangleForm1:  agg1//0, agg2//0     => nft1//0, nft2//0     @ktangfor
198     rule tangleForm2:  agg1//0, nft1//0     => nft1//0, nft2//0     @ktangfor
199 }
200
201 #rule 17–19
202 rule ProteasomeInhibitionAggTau(agg: AggTau, prot: Proteasome):
203     agg//0, prot//0 => agg.y+prot.x                                 @kinhibprot
204
205 complex(gsk: GSK3b, p53: P53, abeta: Abeta){
206     rule AbetaProductionU:
207         gsk.x+p53.b(u) => gsk.x+p53.b(u), abeta                     @kprodAbeta
208
209     rule AbetaProductionP:
210         gsk.x+p53.b(p)  => gsk.x+p53.b(p), abeta                    @kprodAbeta
211
212     rule AbetaDegradation:  abeta//0        => _                    @kprodAbeta
213 }
214
215 #abeta aggregation ____ rule 20
216 complex(abeta1: Abeta, abeta2: Abeta, agg1: AggAbeta, agg2: AggAbeta){
217     rule AbetaAgg1:
218         abeta1//0, abeta2//0    => agg1//0                          @kaggAbeta/2
219
220     rule AbetaAgg2:
221         abeta1//0, agg1.y//0    => agg1.y//0, agg2.y//0             @kaggAbeta
222 }
223
224 #abeta plaque formation ____ rule 21
225 complex(agg1: AggAbeta, agg2: AggAbeta, plaque1: AbetaPlaque, plaque2: AbetaPlaque){
226     rule AbetaPlaque1:
227         agg1//0, agg2//0    => plaque1//0, plaque2//0               @kpf/2
228
229     rule AbetaPlaque2:
230         agg1//0, plaque1//0 => plaque1//0, plaque2//0               @kpf
231 }
232
233 # rules 22–23
234 rule ProteasomeInhibitionAggAbeta(agg: AggAbeta, prot: Proteasome):
235     agg//0, prot//0 => agg.y+prot.x                                 @kinhibprot
236
237 rule P53Transcription(abeta: Abeta, p53mRNA: P53mRNA):
238     abeta//0        => abeta//0, p53mRNA//0                         @ksynp53mRNAAbeta
239
240 # AbetaRosProduction ____ rule 24
241 complex (agg: AggAbeta, ros: ROS){
242     rule AbetaROSProduction1:   agg//0 => agg//0, ros//0            @kgenROSAbeta
243     rule AbetaRosProduction2:
244         agg.y+Proteasome.x => agg.y+Proteasome.x, ros//0            @kgenROSAbeta
245 }
246
247 #rules 25–27
248 rule P53Synthesis(p53: P53, p53mRNA: P53mRNA):  p53mRNA//0
249      => p53mRNA//0, p53.b(u)//0                                     @ksynp53
250
251 rule P53_Mdm2(p53: P53, mdm2: Mdm2): p53.b(u)//0, mdm2.z(u)//0,mdm2.ubi(unb)//0
252     <=> p53.b(u)+mdm2.z(u),mdm2.ubi(unb)//0                         @kbinMdm2p53,krelMdm2p53
253
254 rule E1Ub(e1: E1, ub: Ub, atp: ATP, amp: AMP):
255     ub, e1.x(unb)//0, atp       => e1.x(ubi)//0, amp                @kbinE1Ub/5000
256
257 rule E2Ub(e1: E1, e2: E2):
258     e1.x(ubi)//0, e2.x(unb)//0  => e2.x(ubi)//0, e1.x(unb)//0       @binE2Ub
259
260
261
```

```
#PolyUbiquination ____ rules 28–30
complex(mdm2: Mdm2, p53: P53, e2: E2){
    rule P53Ubiquitination: mdm2.z(u)+p53.b(u), mdm2.ubi(unb)//0, e2.x(ubi)//0
        => mdm2.z(u)+p53.b(ub1), mdm2.ubi(unb)//0, e2.x(unb)//0      @kp53Ub

    rule P53PolyUbiquitination1: mdm2.z(u)+p53.b(ub1), mdm2.ubi(unb)//0, e2.x(ubi)//0
        => mdm2.z(u)+p53.b(ub2), mdm2.ubi(unb)//0, e2.x(unb)//0      @kp53PolyUb

    rule P53PolyUbiquitination2:    mdm2.z(u)+p53.b(ub2), mdm2.ubi(unb)//0, e2.x(ubi)//0
        => mdm2.z(u)+p53.b(ub3), mdm2.ubi(unb)//0, e2.x(unb)//0      @kp53PolyUb

    rule P53PolyUbiquitination3:    mdm2.z(u)+p53.b(ub3), mdm2.ubi(unb)//0, e2.x(ubi)//0
        => mdm2.z(u)+p53.b(ub4), mdm2.ubi(unb)//0, e2.x(unb)//0      @kp53PolyUb
}

#PolyDeUbiquination ____ rules 28–30
complex(mdm2: Mdm2, p53: P53, p53dub: P53DUB, ub: Ub){
    rule P53DeUbiquitination:       mdm2.z(u)+p53.b(ub1), mdm2.ubi(unb)//0, p53dub
        => mdm2.z(u)+p53.b(u),   p53dub, mdm2.ubi(unb)//0, ub        @kactDUBp53

    rule P53PolyDeUbiquitination2:  mdm2.z(u)+p53.b(ub2), mdm2.ubi(unb)//0, p53dub
        => mdm2.z(u)+p53.b(ub1), p53dub, mdm2.ubi(unb)//0, ub        @kactDUBp53

    rule P53PolyDeUbiquitination3:  mdm2.z(u)+p53.b(ub3), mdm2.ubi(unb)//0, p53dub
        => mdm2.z(u)+p53.b(ub2), p53dub, mdm2.ubi(unb)//0, ub        @kactDUBp53

    rule P53PolyDeUbiquitination4:  mdm2.z(u)+p53.b(ub4), mdm2.ubi(unb)//0, p53dub
        => mdm2.z(u)+p53.b(ub3), p53dub, mdm2.ubi(unb)//0, ub        @kactDUBp53
}

#rules 31–34
complex(mdm2: Mdm2, p53: P53, prot: Proteasome){
    rule P53ProteasomeBinding: mdm2.z(p)+p53.b(ub4), prot//0
        => mdm2.z(u)//0, p53.b(ub4)+prot.x                           @kbinProt

    rule P53Degradation(atp: ATP, adp: ADP, ub1: Ub, ub2: Ub, ub3: Ub, ub4: Ub):
        p53.b(ub4)+prot.x, atp
            => adp, ub1//0, ub2//0, ub3//0, ub4//0, prot.x//0        @kdegp53/5000


    complex(mdm2mRna: Mdm2mRNA){
        rule mdm2mRNASynthesis:      p53.b//0
            => p53.b//0, mdm2mRna//0                                 @ksynMdm2mRNA

        rule mdm2mRNADegradation:    mdm2mRna//0 => _                @kdegMdm2mRNA

        #rule 35
        rule Mdm2Synthesis:    mdm2mRna//0 => mdm2mRna//0, mdm2//0   @ksynMdm2
    }
}

complex(mdm2: Mdm2, e2: E2){
    rule mdm2UbiquinationU:      mdm2.ubi(unb)//0, mdm2.z(u)//0, e2.x(ubi)//0
        => mdm2.ubi(ub1)//0, mdm2.z(u)//0, e2.x(unb)//0              @kMdm2Ub

    rule mdm2UbiquinationP:      mdm2.ubi(unb)//0, mdm2.z(p)//0, e2.x(ubi)//0
        => mdm2.ubi(ub1)//0, mdm2.z(p)//0, e2.x(unb)//0              @kMdm2PUb

    rule mdm2PolyUbiquination1: mdm2.ubi(ub1)//0, e2.x(ubi)//0
        => mdm2.ubi(ub2)//0, e2.x(unb)//0                            @kMdm2PolyUb

    rule mdm2PolyUbiquination2: mdm2.ubi(ub2)//0, e2.x(ubi)//0
        => mdm2.ubi(ub3)//0, e2.x(unb)//0                            @kMdm2PolyUb

    rule mdm2PolyUbiquination3: mdm2.ubi(ub3)//0, e2.x(ubi)//0
        => mdm2.ubi(ub4)//0, e2.x(unb)//0                            @kMdm2PolyUb

```

```
329     complex(mdm2dub: Mdm2DUB, ub: Ub){
330         rule mdm2deubiquination:        mdm2.ubi(ub1)//0, mdm2dub
331             => mdm2.ubi(unb)//0, mdm2dub, ub                        @kactDUBMdm2
332
333         rule mdm2polyDeubiquination1:   mdm2.ubi(ub2)//0, mdm2dub
334             => mdm2.ubi(ub1)//0, mdm2dub, ub                        @kactDUBMdm2
335
336         rule mdm2polyDeubiquination2:   mdm2.ubi(ub3)//0, mdm2dub
337             => mdm2.ubi(ub2)//0, mdm2dub, ub                        @kactDUBMdm2
338
339         rule mdm2polyDeubiquination3:   mdm2.ubi(ub4)//0, mdm2dub
340             => mdm2.ubi(ub3)//0, mdm2dub, ub                        @kactDUBMdm2
341     }
342 }
343
344 complex(mdm2: Mdm2, prot: Proteasome, adp: ADP, atp: ATP){
345     rule Mdm2ProteasomeBind:
346         mdm2.ubi(ub4)//0, prot//0 => mdm2.ubi(ub4)+prot.x           @kbinProt
347
348     rule Mdm2Degradation(ub1: Ub, ub2: Ub, ub3: Ub, ub4: Ub):
349         prot.x+mdm2.ubi(ub4), atp =>
350         ub1//0, ub2//0, ub3//0, ub4//0, prot.x//mdm2.ubi(ub4), adp  @kdegMdm2/5000
351 }
352
353 #rule 36-37
354 complex(atma:ATMA, atmi: ATMI, damDNA: DamDNA){
355     rule ATMActivation:     damDNA, atmi    => damDNA, atma         @kactATM
356     rule ATMInactivation:   atma            => atmi                 @kinactATM
357 }
358
359 #rule 38-39
360 complex(p53: P53, atma: ATMA, mdm2: Mdm2){
361     rule P53Phospho:    p53.b(u)//0, atma   => p53.b(p)//0, atma    @kphosp53
362     rule P53DePhospho:  p53.b(p)//0         => p53.b(u)//0          @kdephosp53
363     rule Mdm2Phospho:   mdm2.z(u)//0, mdm2.ubi(unb)//0, atma
364         => mdm2.z(p)//0, mdm2.ubi(unb)//0, atma                     @kphosMdm2
365
366     rule Mdm2DePhospho: mdm2.z(p)//0, mdm2.ubi(unb)//0
367         => mdm2.z(u)//0, mdm2.ubi(unb)//0                           @kdephosMdm2
368 }
369
370 #rule 40-41
371 complex(p53mRna: P53mRNA){
372     rule p53mRNASynthesis:      _           => p53mRna              @ksynp53mRNA
373     rule p53mRNADegradation:    p53mRna => _                        @kdegp53mRNA
374 }
375
376 #DNA repair and damage ____ rules 42-44
377 complex(damDNA: DamDNA, ir: IR, ros: ROS, basalRos: BasalROS){
378     rule IRDamage:          ir => ir, damDNA                        @kdam
379     rule DNARepair:         damDNA => _                             @krepair
380     rule ROSDamage:         ros => ros, damDNA                      @kdamROS
381     rule BasalROSDamage:    basalRos => basalRos, damDNA            @kdamBasalROS
382 }
383
384 #Observers
385 observe mdm2(mdm2: Mdm2):                       mdm2
386 observe mdm2mRNA(mdm2mRna: Mdm2mRNA):           mdm2mRna
387 observe p53(p53: P53):                          p53
388 observe p53mRNA(p53mRna: P53mRNA):              p53mRna
389 observe p53gsk(p53: P53, gsk: GSK3b):           p53.b+gsk.x
390 observe tauTangle(nft: NFT):                    nft
391 observe abetaPlaque(abetaPlaque: AbetaPlaque):  abetaPlaque
392
393 terminate iterations = 10000
```

Literaturverzeichnis

[1] P. Boutillier, J. Feret, J. Krivine, and W. Fontana, *The Kappa Language and Tools*. , v4 ed., November 2019.

[2] L. A. Harris, J. S. Hogg, J.-J. Tapia, J. A. P. Sekar, S. Gupta, I. Korsunsky, A. Arora, D. Barua, R. P. Sheehan, and J. R. Faeder, "BioNetGen 2.2: advances in rule-based modeling," *Bioinformatics*, vol. 32, pp. 3366–3368, 07 2016.

[3] L. Fritsche and G. Kulcsár, "emoflon: A tool for tools and transformations," in *Modellierung 2018* (I. Schaefer, D. Karagiannis, A. Vogelsang, D. Méndez, and C. Seidl, eds.), (Bonn), pp. 301–302, Gesellschaft für Informatik e.V., 2018.

[4] S. Ehmes, L. Fritsche, and A. Schürr, "Simsg: Rule-based simulation using stochastic graph transformation," *Journal of Object Technology*, vol. 18, pp. 1:1–17, July 2019. The 12th International Conference on Model Transformations.

[5] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, *et al.*, "The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, no. 4, pp. 524–531, 2003.

[6] A. M. Smith, W. Xu, Y. Sun, J. R. Feader, and G. E. Marai, "Rulebender: integrated modeling, simulation and visualization for rule-based intracellular biochemistry," vol. 13, 2012.

[7] A. Funahashi, Y. Matsuoka, A. Jouraku, M. Morohashi, N. Kikuchi, and H. Kitano, "Celldesigner 3.5: a versatile modeling tool for biochemical networks," *Proceedings of the IEEE*, vol. 96, no. 8, pp. 1254–1265, 2008.

[8] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine, "Rule-based modelling of cellular signalling," in *CONCUR 2007 – Concurrency Theory* (L. Caires and V. T. Vasconcelos, eds.), (Berlin, Heidelberg), pp. 17–41, Springer Berlin Heidelberg, 2007.

[9] C. J. Proctor and D. A. Gray, "GSK3 and p53 - is there a link in Alzheimer's disease?," *Molecular Neurodegeneration*, vol. 5, 2010.

[10] C. Hooper, R. Killick, and S. Lovestone, "The GSK3 hypothesis of Alzheimer's disease," *Journal of Neurochemistry*, vol. 104, pp. 1433–1439, 2008.

[11] C. Hooper, E. Meimaridou, M. Tavassoli, G. Melino, S. Lovestone, and R. Killick, "p53 is upregulated in alzheimer's disease and induces tau phosphorylation in hek293a cells," *Neuroscience letters*, vol. 418, no. 1, pp. 34–37, 2007.

[12] D. T. Gillespie, "Exact stochastic simulation of coupled chemical reactions," *The Journal of Physical Chemistry (J. Phys. Chem.)*, vol. 81, pp. 2340–2361, 1977.

[13] W. S. Hlavacek, J. R. Faeder, M. L. Blinov, R. G. Posner, M. Hucka, and W. Fontana, "Rules for Modeling Signal-Transduction Systems," *Science Signaling*, vol. 2006, no. 344, pp. re6–re6, 2006.

[14] J. Faeder, M. Blinov, B. Goldstein, and W. Hlavacek, "Rule-based modeling of biochemical networks: Research Articles," *Complexity*, vol. 10, pp. 22–41, 03 2005.

[15] R. Milner, *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, May 1999.

[16] A. Regev, W. Silverman, and E. Shapiro, "Representation and simulation of biochemical processes using the π-calculus process algebra," in *Biocomputing 2001*, pp. 459–470, 12 2000.

[17] D. T. Gillespie, "A general method for numerically simulating the stochastic time evolution of coupled chemical reactions," *Journal of Computational Physics*, vol. 22, no. 4, pp. 403 – 434, 1976.

[18] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, February 2006.

[19] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice, second edition," *Synthesis Lectures on Software Engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[20] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, pp. 1–17, USA, 2003.

[21] T. Mens and P. V. Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[22] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, "Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework," *Software Systems Modeling*, vol. 15, pp. 609–629, 2016.

[23] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Science of Computer Programming*, vol. 72, pp. 31–39, 06 2008.

[24] D. Eppstein, "Subgraph isomorphism in planar graphs and related problems," in *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, (USA), p. 632–640, Society for Industrial and Applied Mathematics, 1995.

[25] "The eclipse integrated development environment (ide)." . [„`https://www.eclipse.org/eclipseide/`"; last accessed 12-March-2020].

[26] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Trans. Comput. Educ.*, vol. 13, Nov. 2013.

[27] B. A. Myers, J. F. Pane, and A. Ko, "Natural programming languages and environments," *Commun. ACM*, vol. 47, p. 47–52, Sept. 2004.

[28] T. P. McNamara, *Semantic priming: Perspectives from memory and word recognition*. Psychology Press, 2005.

[29] D. D. McCracken and E. D. Reilly, *Backus-Naur Form (BNF)*, p. 129–131. GBR: John Wiley and Sons Ltd., 2003.

[30] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "The scala language specification," 2004.

[31] S. Marlow, "The haskell language report." , 2004. [„`https://www.namsu.de/Extra/pakete/Hyperref.html`"; last accessed 05-March-2020].

[32] E. W. Dijkstra, *On the Role of Scientific Thought*, pp. 60–66. New York, NY: Springer New York, 1982.

[33] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[34] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, (New York, NY, USA), p. 307–309, Association for Computing Machinery, 2010.

[35] "Ebnf: Iso/iec 14977: 1996 (e)," *Standard, EBNF Syntaxt Specification*, vol. 70, 1996. [„`https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`"; last accessed 23-March-2020].

[36] E. International, *ECMA-262: ECMAScript®2018 Language Specification*. , 10th ed., June 2019.

[37] G. Varró and F. Deckwerth, "A rete network construction algorithm for incremental pattern matching," in *Theory and Practice of Model Transformations* (K. Duddy and G. Kappel, eds.), (Berlin, Heidelberg), pp. 125–140, Springer Berlin Heidelberg, 2013.

[38] C. L. Forgy and S. J. Shepard, "Rete: A fast match algorithm," *AI Expert*, vol. 2, p. 34–40, Jan. 1987.

[39] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, p. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[40] A. Goldbeter and D. E. Koshland, "An amplified sensitivity arising from covalent modification in biological systems," *Proceedings of the National Academy of Sciences*, vol. 78, no. 11, pp. 6840–6844, 1981.

[41] M. W. Sneddon, J. R. Faeder, and T. Emonet, "Efficient modeling, simulation and coarse-graining of biological complexity with nfsim," vol. 8, pp. 177–183, 2011.

[42] D. J. Higham and N. J. Higham, *MATLAB guide*, vol. 150. Siam, 2016.

[43] N. Le Novere, M. Hucka, H. Mi, S. Moodie, F. Schreiber, A. Sorokin, E. Demir, K. Wegner, M. I. Aladjem, S. M. Wimalaratne, *et al.*, "The systems biology graphical notation," *Nature biotechnology*, vol. 27, no. 8, p. 735, 2009.