UNIVERSITÄT BIELEFELD

DISSERTATION

Petri-Netz-basierte Simulation biologischer Prozesse mit OpenModelica

Autor:
Lennart A. Ochel

Betreuer:
Prof. Dr. Ralf Hofestädt
Prof. Dr. Bernhard Bachmann

Dissertation zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
in der

AG Bioinformatik
Technische Fakultät
Universität Bielefeld

Dezember 2016

Lennart A. Ochel

Petri-Netz-basierte Simulation
biologischer Prozesse mit OpenModelica

Dissertation

Technische Fakultät der Universität Bielefeld
AG Bioinformatik, Prof. Dr. Ralf Hofestädt

Fachbereich Ingenieurwissenschaften und Mathematik der Fachhochschule Bielefeld
Angewandte Mathematik, Prof. Dr. Bernhard Bachmann

Gutachter:
Prof. Dr. Ralf Hofestädt, Universität Bielefeld
Prof. Dr. Bernhard Bachmann, Fachhochschule Bielefeld
Prof. Dr. Falk Schreiber, Universität Konstanz

Prüfungsausschuss:
Prof. Dr. Tim W. Nattkemper, Universität Bielefeld
Prof. Dr. Ralf Hofestädt, Universität Bielefeld
Prof. Dr. Bernhard Bachmann, Fachhochschule Bielefeld
Prof. Dr. Falk Schreiber, Universität Konstanz
Dr. Basil Ell, Universität Bielefeld

Gedruckt auf alterungsbeständigem Papier (ISO 9706)

UNIVERSITÄT BIELEFELD

Zusammenfassung

AG Bioinformatik
Technische Fakultät

Petri-Netz-basierte Simulation biologischer Prozesse mit OpenModelica

von Lennart A. Ochel

Biologische Prozesse werden häufig mit Hilfe von Petri-Netzen abgebildet. Dies hat den Vorteil, dass die Prozesse automatisch eine grafische Darstellung erhalten und gleichzeitig durch einen mathematischen Formalismus beschrieben sind. Zudem stellt die Verwendung von Petri-Netzen sicher, dass biologische Randbedingungen eingehalten werden. Zum einen entspricht die Nicht-negativität der Markierungen im Petri-Netz dem biologischen Vorkommen von Konzentrationen bzw. Stoffmengen. Zum anderen ist die Richtung der abgebildeten biologischen Reaktionen durch die Vor- und Nachbereiche der entsprechenden Transitionen im Petri-Netz streng definiert. Ein Nachteil vieler biologischer Simulationsumgebungen ist, dass der zugrundeliegende (Petri-Netz-)Formalismus nicht transparent vorliegt. Dieses Problem löst die Bibliothek PNlib, die den xHPN-Formalismus transparent und freizugänglich auf Basis der Modellierungssprache Modelica bereitstellt. Im Folgenden werden die von mir entwickelten und implementierten Konzepte, Methoden und Algorithmen zusammengefasst.

Meine Weiterentwicklung der PNlib wurde als Version 2.0 veröffentlicht, die die Funktionalität der Version 1.0 enthält und diese Modelica-konform und toolunabhängig bereitstellt. Zudem wurde die Bibliothek um Funktionalitäten zu Flussanalyse und gefaltete Plätzen erweitert, welche für die modellbasierte Kohlenstoffflussanalyse eingesetzt werden. Diese Kombination ergibt einen quelloffenen, transparenten und erweiterbaren Petri-Netz-Simulator für zeitbasierte Petri-Netze.

Um den Petri-Netz-Simulator zu ermöglichen, wurde das OpenModelica-Backend, der mathematische Transformationsprozess, neu strukturiert und implementiert. Insbesondere der Initialisierungsprozess wurde überarbeitet, sodass nun auch Petri-Netz-Modelle verarbeitet und in eine effizient zu lösende Form gebracht werden. Es wurden Lösungsstrategien für unter-, über- und gemischt-bestimmte Initialisierungsprobleme erarbeitet und implementiert. Erst diese Erweiterungen ermöglichen eine effiziente Initialisierung und Simulation komplexer hybrider Modelle, wie sie beim Modellieren von Petri-Netzen mit dem xHPN-Formalismus entstehen.

Anhand eines Teilmodells der Glykolyse wird die praktische Relevanz der erfolgten Arbeiten gezeigt. Hierfür wird das metabolische Netzwerk in ein Petri-Netz transformiert und um gefaltete Plätze erweitert. In der anschließenden Betrachtung der Simulationsergebnisse ist es so möglich den Kohlenstoffffluss, ausgehend von markierten Metaboliten, zu verfolgen und zu analysieren.

BIELEFELD UNIVERSITY

Abstract

Bioinformatics / Medical Informatics Department
Faculty of Technology

Petri net-based simulation of biological processes with OpenModelica

by Lennart A. Ochel

Biological processes are often described using Petri net formalisms. This has several advantages, e.g. the biological process is automatically transformed into a visual representation and the dynamics is described by a solid mathematical formalism. The Petri net formalism natively considers several biological boundaries. Places contain a non-negative marking which correspond with the amount or concentration of substances. Transitions and their pre- and postsets define precisely the reaction flow. A common disadvantage of simulation tools is that the simulation process is undocumented. This is solved by the Modelica library PNlib, which provides a transparent definition of the xHPN formalism. In the following, I summarize the concepts, methods and algorithms I have developed and implemented.

The library PNlib has been developed to the new version 2.0, which includes all the functionality of the original 1.0 release in a Modelica compliant and consequently tool independent way. Furthermore, the library has been extended to new functionalities with respect to flux analysis and folded places. This is especially important for model-based carbon flux analysis. The combination of the library and the open source Modelica environment OpenModelica provides a freely accessible Petri net modelling and simulation environment for time-based Petri nets.

To enable this kind of modelling and simulation environment, the given OpenModelica backend, which contains all the mathematical transformations, had to be restructured and reimplemented. In particular, a new approach of the initialization process has been developed, which comprises handling of under-, over-, and mixed-determined systems. All this enables efficient initialization and simulation of complex hybrid systems as they occur in Petri net models using the xHPN formalism.

The practical benefits of the described work are shown using a sub-model of Glycolysis. Therefore, the metabolic network is first converted into Petri net formalism and then extended with folded places. In the concluding evaluation of the simulation results, the carbon fluxes are traced based on certain labelled metabolites.

Danksagung

Diese Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter im Rahmen der Forschungskooperation „Modellbasierte Realisierung intelligenter Systeme in der Nano- und Biotechnologie“ (kurz MoRitS), die vom Ministerium für Innovation, Wissenschaft und Forschung des Landes Nordrhein-Westfalen (MIWF NRW) gefördert wurde.

Mein besonderer Dank gilt Prof. Bernhard Bachmann und Prof. Ralf Hofestädt, die meine Arbeit ermöglicht und betreut haben. Mir wurden die Freiheit und der Rückhalt gegeben, meine eigenen Ideen zu verwirklichen. Gleichzeitig haben die vielen konstruktiven und inspirierenden Diskussionen dafür gesorgt, dass ich mein Ziel stets im Blick behalten habe. Zudem möchte ich mich bei meinen Kollegen Timo Lask, Patrick Täuber und Christoph Brinkrolf bedanken.

Des Weiteren bedanke ich mich bei Prof. Francesco Casella für den wissenschaftlichen Austausch und seine Hartnäckigkeit, mit der er mich und meine Arbeit begleitet hat. Prof. Peter Fritzson, Adrian Pop und Martin Sjölund möchte ich für die Unterstützung im OpenModelica-Projekt danken.

Meinem Kollegen und „Leidensgenossen“ Christoph Brinkrolf gilt für die mehrjährige Zusammenarbeit und Unterstützung mein besonderer Dank, ohne dem diese Arbeit wahrscheinlich nicht, oder zumindest nicht in dieser Form, zustande gekommen wäre.

Der Arbeitsgruppe CPN, dies sind Prof. Hermann-Josef Kruse, Prof. Bernhard Bachmann, Timo Lask und Sabrina Proß, möchte ich für die lebhaften und häufig launigen Diskussionen rund um die Petri-Netze danken.

Zum Schluss möchte ich noch meiner Familie und meinem privaten Umfeld für die Unterstützung in den vergangenen Monaten und Jahren danken.

Inhaltsverzeichnis

« O mathématiques saintes, puissiez-vous, par votre commerce perpétuel, consoler le reste de mes jours de la méchanceté de l'homme et de l'injustice du Grand-Tout! »

Lautréamont, 1869
Les Chants de Maldoror, II. 10

1 Einleitung

In der klassischen Biologie wird versucht, Hypothesen durch die Durchführung von Experimenten zu widerlegen, um neue Erkenntnisse zu erlangen. Durch Verfeinerung der Hypothesen und Experimente wird ein immer tieferes Verständnis des zu untersuchenden Sachverhalts gewonnen. In der Systembiologie hingegen wird versucht, ein komplexes System als Ganzes zu beschreiben. Hierfür werden systemtheoretische Ansätze auf biologische Prozesse angewendet. Dieses erweitert das Vorgehen der klassischen Biologie um die Modellbildung, Simulation und Analyse. Hierfür müssen einzelne biologische Modelle, die häufig als biologisches Netzwerk vorliegen, in mathematische Modelle überführt und verbunden werden. Zu diesem Zweck gibt es unterschiedliche Ansätze, die von gewöhnlichen Differentialgleichungssystemen bis hin zu zeitbasierten Petri-Netzen reichen.

Petri-Netze verfügen über eine kanonische grafische Darstellung, die vergleichbar mit biologischen Netzen ist, und daher sowohl von Mathematikern als auch von Biologen interpretiert werden kann. Die Vorteile von Petri-Netzen sind insbesondere der strikte mathematische Formalismus und die Möglichkeit nebenläufige Prozesse abzubilden. Zudem gibt es zahlreiche Erweiterungen des Petri-Netz-Formalismus, wie z.B. funktionale Petri-Netze, kapazitive Petri-Netze und gefärbte Petri-Netze. Alle diese Konzepte lassen sich sinnvoll verbinden und ermöglichen so eine einheitliche Beschreibung komplexer Prozesse.

Obwohl es bereits verschiedene mathematische Formalsimen für unterschiedliche Petri-Netz-Klassen gibt, existiert kein freizugänglicher Petri-Netz-Simulator auf Basis eines solchen mathematischen Formalismus. In der Dissertation von Sabrina Proß [1] wurden einige Petri-Netz-Klassen verbunden und mathematisch als xHPN-Formalismus beschrieben. Zudem wurde auf Basis der Modellierungssprache Modelica die Bibliothek PNlib entwickelt, mit der auf Grundlage des xHPN-Formalismus Petri-Netze erstellt werden können. Unter Verwendung der kommerziellen Modelica-Umgebung Dymola (Version 2012 FD01) können diese Modelle auch simuliert werden.

1.1 Ziel der Arbeit

Aufbauend auf der Petri-Netz-Bibliothek PNlib von Sabrina Proß wird eine Modelica-konforme, und somit toolunabhängige, Bibliothek entwickelt, die den xHPN-Formalismus abdeckt. Basierend auf dem frei verfügbaren Modelica-Compiler OpenModelica wird so eine kostenfreie Möglichkeit zur Modellierung und Simulation von zeitbasierten Petri-Netzen geschaffen. Dieser freie Petri-Netz-Simulator wird in das Werkzeug VANESA integriert, um die Modellierung speziell von biologischen Netzwerken zu unterstützen und Simulationsergebnisse für diese Netzwerke bereitzustellen.

Aus diesem Ziel ergeben sich folgende vier konkrete Aufgabestellungen:

Weiterentwicklung der Modelica-Bibliothek PNlib

Die Bibliothek PNlib 1.0 ermöglicht das Modellieren und Simulieren von Petri-Netzen gemäß des xHPN-Formalismus unter Verwendung der damaligen Version (2012 FD01) des kommerziellen Werkzeugs Dymola[1]. Zunächst wird die Bibliothek soweit überarbeitet, dass sie dem Modelica-Sprachstandard entspricht und so nicht mehr an einen speziellen Modelica-Compiler gebunden ist.

Zudem wird die Bibliothek PNlib um gefaltete Plätze erweitert. Hierfür wird ein mathematischer Formalismus entwickelt und dieser anschließend in die Modelica-Bibliothek integriert.

Umstrukturierung des OpenModelica-Backend

In der derzeitigen Version ist eine effiziente Verarbeitung und Aufbereitung hybrider differential-algebraischer Gleichungssysteme nicht möglich. Für die Verarbeitung anspruchsvoller Modelle mit dem OpenModelica-Backend ist eine Umstrukturierung erforderlich. Hierfür wird zunächst der Istzustand analysiert und auf Basis dessen ein neues Konzept entwickelt und umgesetzt.

Initialisierung gemischter Systeme in OpenModelica

Zu Beginn einer jeden Simulation muss zunächst ein konsistenter Startzustand berechnet werden. Dieser Prozess nennt sich Initialisierung und ist häufig Knackpunkt der gesamten Simulation. Petri-Netz-basierte Modelle enthalten einen hohen Anteil an diskreten Elementen, die in diesem Prozess berücksichtigt werden müssen. Dies gilt auch für kontinuierliche Petri-Netze, da auch hier Aktivierungs- und Feuerprozesse diskrete Anteile enthalten. Diese Modelle lassen sich aufgrund der komplexen Mischung von kontinuierlichen und diskreten Elementen derzeit nicht mit OpenModelica initialisieren und dadurch auch nicht simulieren. Hierfür wird die Initialisierung auf Basis symbolischer Vorbehandlung neuentwickelt und implementiert.

Petri-Netz-basierte Simulation von Kohlenstoffflüssen innerhalb eines biologischen Netzes

Die praktische Relevanz wird an einem Netzwerk der Glykolyse demonstriert, bei dem die Kohlenstoffatome eines bestimmten Ausgangsmetaboliten markiert werden. Diese können dann während der Simulation im gesamten Netzwerk verfolgt werden. Dadurch lassen sich Wege und Anreicherungen dieser Atome im Netzwerk analysieren.

---

[1] Dymola ist eine kommerzielle Modelica-Umgebung:
⇒ http://www.3ds.com/products-services/catia/products/dymola

1.2 Aufbau der Arbeit

Im folgenden Kapitel werden relevante Grundlagen eingeführt. Dies umfasst die Modellierungssprache Modelica, Petri-Netze und die Modellierung von Reaktionskinetiken. In dem Modelica-Teil werden grundlegende Sprachelemente eingeführt. Danach wird gezeigt, wie Modelica aufgrund diverser Bibliotheken, z.B. der PNlib, zu einem Werkzeug unterschiedlichster Disziplinen geworden ist. Schließlich wird auf die freie Modelica-Umgebung OpenModelica näher eingegangen. Danach werden Petri-Netze eingeführt und einige grundlegende Konzepte definiert. Im letzten Abschnitt des Kapitels „Grundlagen“ wird die Modellierung von Reaktionskinetiken anhand unterschiedlicher Ansätze gezeigt.

In Kapitel 3 „Verwandte Arbeiten“ werden bestehende Werkzeuge für die Simulation von biologischen Netzen und insbesondere auch Petri-Netzen vorgestellt und verglichen.

Kapitel 4 „Neuentwurf und Implementierung des OpenModelica-Backend“ setzt sich intensiv mit dem Ausgangszustand des OpenModelica-Backend und den angestrebten strukturellen Änderungen auseinander. Es werden zudem Algorithmen zur Lösung von Initialisierungsproblemen hybrider differentialalgebraischer Gleichungssysteme vorgestellt.

Kapitel 5 „Weiterentwicklung der PNlib“ beschreibt die Arbeit an der Modelica-Bibliothek PNlib. Zu Beginn des Kapitels wird die Testumgebung beschrieben. Anschließend wird auf die Erweiterung der Bibliothek, und insbesondere auf gefärbte Plätze, eingegangen.

In Kapitel 6 „Anwendung: Glykolyse in Saccharomyces cerevisiae“ werden die zuvor eingeführten Petri-Netz-Konzepte zusammen mit den Arbeiten an dem OpenModelica-Compiler anhand eines metabolischen Teilsystems der Glykolyse demonstriert.

Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick in Kapitel 7.

2 Grundlagen

2.1 Modelica

Modelica ist eine Sprache für die Modellierung von dynamischen Systemen, welche von der Modelica Association entwickelt wurde und weiterentwickelt wird. Die Modelica Association ist eine gemeinnützige Organisation, mit Mitgliedern aus Europa, USA, Canada und Asien. Seit 1996 arbeitetet sie an der offenen Sprache Modelica und der zugehörigen quelloffenen Standardbibliothek, der Modelica Standard Library (kurz MSL). (vgl. [2])

Mit Modelica wird das Ziel verfolgt, technische Systeme aus den unterschiedlichsten Gebieten, wie z.B. der Mechanik, Elektrotechnik, Pneumatik und Regelungstechnik, einfach und komfortabel zu beschreiben und miteinander zu verbinden. Hierzu werden die Modelle durch algebraische Gleichungen, Differentialgleichungen und diskrete Gleichungen beschrieben. (vgl. [3])

Modelica findet etwa seit dem Jahr 2000 Einsatz in der Industrie. Dies spiegelt sich auch in der Mitgliederliste der Modelica Association, mit Firmen wie BMW, Dassault Systèmes, Maplesoft, Siemens und vielen weiteren, wieder.

Die nachstehende Tabelle führt einige der wichtigen Modelica-Umgebungen[1] auf.

Tabelle 2.1: Liste ausgewählter Modelica-Umgebungen (in alphabetischer Reihenfolge).

Name	Entwickler	Lizenz
Dymola	Dassault Systèmes AB	kommerziell
JModelica.org	Modelon AB	frei
MapleSim	Maplesoft	kommerziell
OpenModelica	Open Source Modelica Consortium (OSMC)	frei
SimulationX	ESI ITI GmbH	kommerziell
SystemModeler	Wolfram MathCore AB	kommerziell

2.1.1 Modelica Sprachelemente

Für gewöhnlich werden Modelica-Modelle aus unterschiedlichen Komponenten hierarchisch und objekt-orientiert aufgebaut. Aufgrund des gleichungsbasierten Ansatzes der Sprache können Komponenten auf einer physikalisch-mathematischen Ebene beschrieben werden, ohne bereits die Kausalitäten der späteren Anwendung berücksichtigen zu müssen. Dies wird bereits an einem einfachen Modell, wie dem des elektrischen Widerstandes deutlich, der im einfachsten Fall durch das Ohm'sche Gesetz $u = R \cdot i$ beschrieben werden kann. Hierbei ist R der charakteristische Parameter des Widerstandes und u und i die Variablen für den elektrischen Spannungsabfall sowie den elektrischen Strom. Eine der beiden Variablen muss

[1] Eine aktuelle Liste der verfügbaren Modelica-Programme stellt die Modelica Association auf ihrer Webseite bereit: https://www.modelica.org/tools.

bekannt sein, sodass die jeweils andere mit Hilfe des gerade beschriebenen Ohm'schen Gesetzes berechnet werden kann. Die Kausalität, also ob die Spannung oder der Strom im späteren Modell als Unbekannte auftritt, ist hierbei für die Modellierung des Widerstandes unerheblich. Dies unterscheidet Modelica grundlegend von anderen Modellierungssprachen bzw. Modellierungswerkzeugen auf Basis gerichteter Flüsse, wie etwa Matlab Simulink[1] und stellt einen großen Vorteil dar.

Modelica-Modelle können durch die Definition von Variablen und Gleichungen, sowie durch die Kombination bestehender Modelle erstellt werden. Es besteht die Möglichkeit, diese beiden Techniken frei zu kombinieren. Einzelne Modelle können sowohl durch kausale als auch akausale Verbindungen miteinander verbunden werden. Für die akausalen Verbindungen werden spezielle Konnektoren für die jeweiligen Schnittstellen definiert, die auf dem Prinzip von Fluss- und Potentialvariablen beruhen [4]. Die Idee hierbei ist, dass von jeder Verbindung automatisch Gleichungen abgeleitet werden können, die das Zusammenspiel der verbundenen Objekte beschreiben. Ein elektrischer Konnektor besteht z.B. aus einer Potentialvariable für die elektrische Spannung und einer Flussvariable für den elektrischen Strom. Ein Konnektor zur Modellierung von Stoffkonzentrationsänderungen auf Basis von Enzymreaktionen könnte z.B. aus einer Potentialvariable für die Stoffkonzentration und einer Flussvariable für den Stoffmengendurchfluss aufgebaut werden. Die automatisch generierten Gleichungen werden durch das Gleichsetzen aller Potentialvariablen innerhalb einer Verbindung und durch die zusätzliche Bedingung, dass die Summe aller Flussvariablen einer Verbindung 0 ist, gebildet.

Darüber hinaus gibt es noch weitere spezielle Modell-Klassen, wie z.B. Funktionen und Blöcke. Bei diesen beiden Klassen werden die Eingangs- und Ausgangsgrößen explizit definiert.

Modelica basiert auf einem strikten Determinismus. Dies bedeutet, dass Funktionsaufrufe mit gleichen Eingabegrößen immer die gleichen Ausgabegrößen berechnen müssen. Mit der Version 3.3 des Modelica-Sprachstandards wurde eine Möglichkeit eingeführt diese Eigenschaft explizit für einzelne Funktionen aufzuheben, um z.B. einen Zufallszahlengenerator formulieren zu können, der bei jedem Aufruf einen neuen Wert zurückgibt.

An dieser Stelle sei für eine ausführliche Beschreibung der Modellierungssprache Modelica auf [5], [6], [7] verwiesen.

2.1.2 Modelica Bibliotheken

Eine weitere Stärke von Modelica ist die umfangreiche Standardbibliothek, die kostenfrei genutzt werden kann. Die Bibliothek umfasst viele unterschiedliche Fachgebiete, wie die Tabelle 2.2 zeigt. Sie wird kontinuierliche weiterentwickelt und erschien im April 2016 in der derzeitigen aktuellen Version 3.2.2.

[1] Für weiter Informationen zu Matlab Simulink siehe http://de.mathworks.com/products/simulink/.

Tabelle 2.2: Aufbau und Versionsverlauf der Modelica Standard Bibliothek.

Aufbau	*Datum*	*Version*
Modelica	Juni 2001	Version 1.4
User's Guide	Dezember 2002	Version 1.5
Blocks	Juni 2004	Version 1.6
ComplexBlocks	November 2004	Version 2.1
StateGraph	April 2005	Version 2.2
Electrical	März 2006	Version 2.2.1
Magnetic	August 2007	Version 2.2.2
Mechanics	März 2008	Version 3.0
Fluid	Januar 2009	Version 3.0.1
Media	August 2009	Version 3.1
Thermal	Oktober 2010	Version 3.2
Math	August 2013	Version 3.2.1
ComplexMath	April 2016	Version 3.2.2
Utilities		
Constants		
Icons		
SIunits		

Neben der Standardbibliothek gibt es zahlreiche weitere Modelica-Bibliotheken für diverse Problemstellungen. Eine umfassende Liste von Modelica-Bibliotheken ist auf der Modelica-Webseite[1] aufgeführt. An dieser Stelle sei noch explizit auf die beiden Bibliotheken *PNlib* und *BioChem* hingewiesen, die in dem Kapitel „Verwandte Arbeiten“ ausführlich vorgestellt werden. Die Bibliothek PNlib stellt alle Komponenten für die Modellierung von Petri-Netzen auf Basis des xHPN-Formalismus bereit, mit denen z.B. metabolische Prozesse abgebildet werden können (vgl. [1]). Die Bibliothek BioChem erlaubt hingegen die direkte Modellierung von biologischen Netzwerken (vgl. [8]).

2.1.3 OpenModelica

OpenModelica[2] ist eine quelloffene Modelica-basierende Modellierungs- und Simulationsumgebung für industrielle und akademische Anwendungen. Es wird von dem *Open Source Modelica Consortium* (kurz OSMC) entwickelt. Das OSMC besteht aus einer wachsenden Gruppe von Firmen, Instituten und Einzelpersonen. Die Universität Linköping übernimmt hierbei die Leitung und einen Großteil der Entwicklungsarbeit. Ziel des OpenModelica-Projektes ist es, eine Referenzimplementierung für den Modelica-Sprachstandard bereitzustellen. Herzstück des Projektes ist der OpenModelica-Compiler, der Modelica Quelltext verarbeiten und ausführbaren Simulationscode erzeugen kann. Darüber hinaus umfasst das OpenModelica-Projekt eine Reihe von Programmen mit grafischer Benutzeroberfläche (z.B. OMEdit und OMNotebook), die ihrerseits den OpenModelica-Compiler aufrufen. Damit stellt dieses Projekt alle Komponenten, die für die Arbeit mit Modelica notwendig sind, kostenfrei zur Verfügung. Das Zusammenspiel der einzelnen

[1] Aktuelle Liste der verfügbaren Modelica-Bibliotheken: https://modelica.org/libraries.
[2] Die OpenModelica-Webseite: https://www.openmodelica.org/.

Komponenten ist schematisch in Abbildung 2.1 dargestellt. Weitere Informationen können [9] entnommen werden.

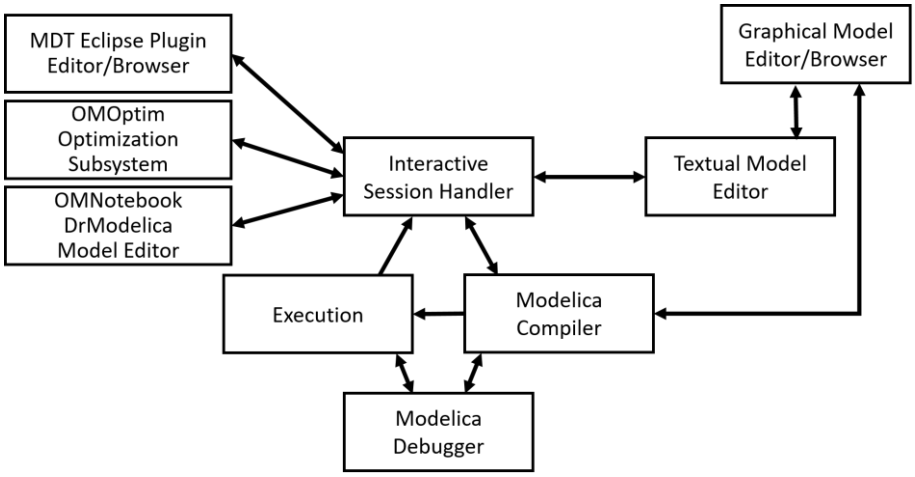

Abbildung 2.1: Darstellung der OpenModelica-Struktur. Pfeile deuten Daten- und Kontrollflüsse an. Der „Interactive Session Handler" empfängt Anweisungen und gibt die Ergebnisse von deren Ausführung zurück. Unterschiedliche Teilsysteme stellen verschiedene Möglichkeiten bereit, um mit Modelica-Code zu arbeiten. (vgl. [9])

2.1.3.1 OpenModelica Compiler

Der OpenModelica Compiler kann, wie bei Compilern üblich, in Frontend, Backend und Codegenerierung unterteilt werden. Eine stark vereinfachte Übersicht ist in Abbildung 2.2 dargestellt.

Das Frontend verarbeitet den Modelica Quelltext und unterzieht diesen einer lexikalischen, syntaktischen und semantischen Analyse. Anschließend wird das analysierte Modell ausgeflacht. Dies bedeutet, dass ein äquivalentes Modell mit allen Variablen und Gleichungen, jedoch ohne hierarchische Struktur, erstellt wird. Dieses Modell wird daher als *flaches Modell* bezeichnet. Das flache Modell wird in einem Zwischenschritt an das Backend übergeben.

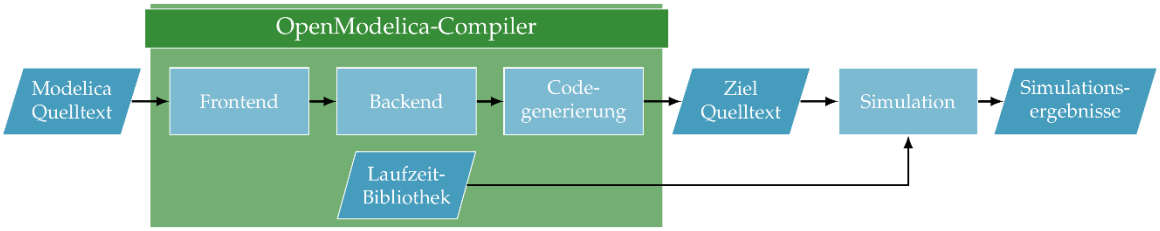

Abbildung 2.2: Vereinfachter Ablauf vom Modelica Modell bis zum Simulationsergebnis. Der Modelica-Quelltext wird vom OpenModelica-Compiler in den Ziel-Quelltext (standardmäßig C) übersetzt und gemeinsam mit der Laufzeitbibliothek zur Simulation kompiliert. Diese erzeugt anschließend die Simulationsergebnisse.

Das Backend verarbeitet das flache Modell und wendet hierzu eine Reihe von Modulen an, um dieses soweit zu verarbeiten, dass daraus der Simulationscode erzeugt werden kann. Auf Aufbau und Funktionsweise wird im nächsten Abschnitt näher eingegangen.

Der generierte Simulationscode wird gemeinsam mit der zugehörigen Laufzeit-Bibliothek zur eigentlichen Simulation kompiliert. Die so erzeugte Simulation kann mit einer Vielzahl von

Parametern konfiguriert werden und berechnet die Simulationsergebnisse für ein definiertes Zeitfenster. Hierauf wird im Abschnitt Laufzeitumgebung näher eingegangen.

2.1.3.1.1 Backend

Das OpenModelica-Backend ist für die Verarbeitung ausgehend vom flachen Modells bis hin zur Codegenerierung zuständig. Wie in Abbildung 2.3 dargestellt, umfasst dies im Wesentlichen drei Verarbeitungsschritte:

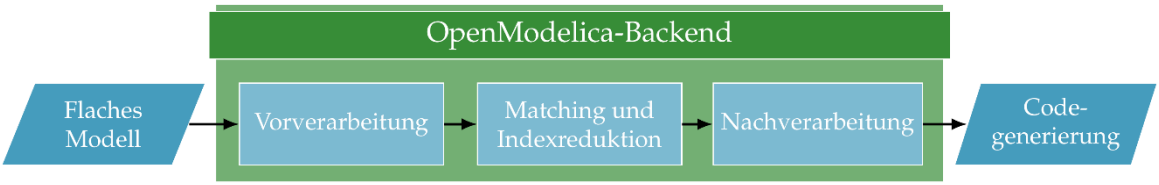

Abbildung 2.3: Übersicht des OpenModelica-Backend mit den drei Phasen „Vorverarbeitung“, „Matching und Indexreduktion“ und „Nachverarbeitung“.

Die symbolische **Vorverarbeitung** der Gleichungen umfasst eine Reihe von Modulen, die auf dem Gleichungssystem arbeiten und keine Matching-Informationen benötigen. Dies sind z.B. Vereinfachungen von bestimmten Funktionsausdrücken und das Aufspüren sogenannter einfacher Gleichungen, also Gleichungen der Form $a = b$, $a = -b$, $a + b = 0$ und so weiter.

Anschließend werden **Matching und Indexreduktion** aufgerufen. Das Matching (siehe [10]) ordnet jeder Gleichung eine Variable zu. Ist dies nicht möglich, spricht man von einem strukturell singulären System. Ursache hierfür ist entweder ein Modellierungsfehler oder ein sogenanntes höheres Index-Problem. Dies bezeichnet DAE-Systeme mit einem differentiellen Index (siehe [11, p. 17]) größer als 1. In diesen Fällen wird versucht, das System durch die sogenannte Indexreduktion (siehe [12], [13]) in ein System mit Index 1 zu transformieren. Für dieses existiert dann wiederum ein perfektes Matching.

Anschließend folgt die symbolische **Nachverarbeitung** des Systems. Dies umfasst erneut eine Reihe von Modulen, die im Gegensatz zur Vorverarbeitung Zugriff auf die Matching-Informationen des Gleichungssystems und die Zustandswahl haben. Dies erlaubt weitere Optimierungen des Gleichungssystems, z.B. durch Tearing [14].

2.1.3.1.2 Laufzeitumgebung

Der Begriff „Simulation“ wird in verschiedenen Zusammenhängen mit unterschiedlichen Bedeutungen verwendet. Im Folgenden beschreibt „Simulation“ das Zusammenspiel aller Phasen, die notwendig sind, um das dynamische Verhalten eines gemischten Systems (insbesondere auch rein kontinuierlicher, respektive diskreter Systeme) zu berechnen. Die nachfolgende Grafik zeigt das Zusammenspiel dieser Phasen.

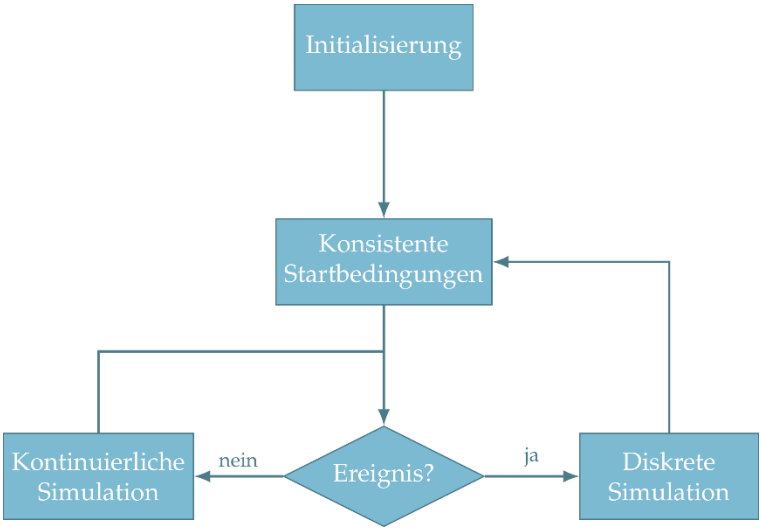

Abbildung 2.4: Zusammenspiel der Phasen zur Berechnung des dynamischen Systemverhaltens hybrider Systeme.

Die Simulation beginnt immer mit der Initialisierung. In dieser Phase werden alle Modellgrößen zum ersten Mal berechnet. Daran anschließend folgt die Berechnung konsistenter Start-Bedingungen. In dieser Phase werden konsistente Bedingungen ($v = pre(v)$) für alle, insbesondere für diskrete, Variablen bestimmt. Dies ist notwendig, um einen konsistenten Systemzustand im Falle aktiver diskreter Gleichungen zu gewährleisten. Ausgehend von einem solchen konsistenten Zustand wird geprüft, ob ein Ereignis vorliegt. Abhängig davon wird mit einem kontinuierlichen bzw. diskreten Simulationsschritt fortgefahren. Nach einem diskreten Simulationsschritt folgt wieder das Bestimmen konsistenter Bedingungen. Nach einem kontinuierlichen Simulationsschritt ist dies automatisch gewährleistet, wodurch direkt zur Überprüfung auf neue Ereignisse zurückgekehrt wird.

2.1.3.2 OMEdit

OMEdit [9, pp. 25-44] ist die zentrale grafische Benutzeroberfläche für die Modellierung und Simulation mit OpenModelica. Diese erlaubt das Durchsuchen von Bibliotheken, Erstellen neuer Modelle, Simulieren und Betrachten von Ergebnissen. Modelle können sowohl durch Drag & Drop bereits erstellter Modelle als auch durch das Bearbeiten der Modelle auf textueller Ebene erstellt und bearbeitet werden.

Die grafische Oberfläche ist für eine Vielzahl von Disziplinen und Bibliotheken, wie z.B. der MSL, gut geeignet. Jedoch lassen sich Petri-Netze mit der PNlib innerhalb dieser Umgebung nicht komfortabel erstellen und bearbeiten. Daher ist speziell für Petri-Netze eine andere Oberfläche, z.B. VANESA (siehe Abschnitt 3.3), zu empfehlen.

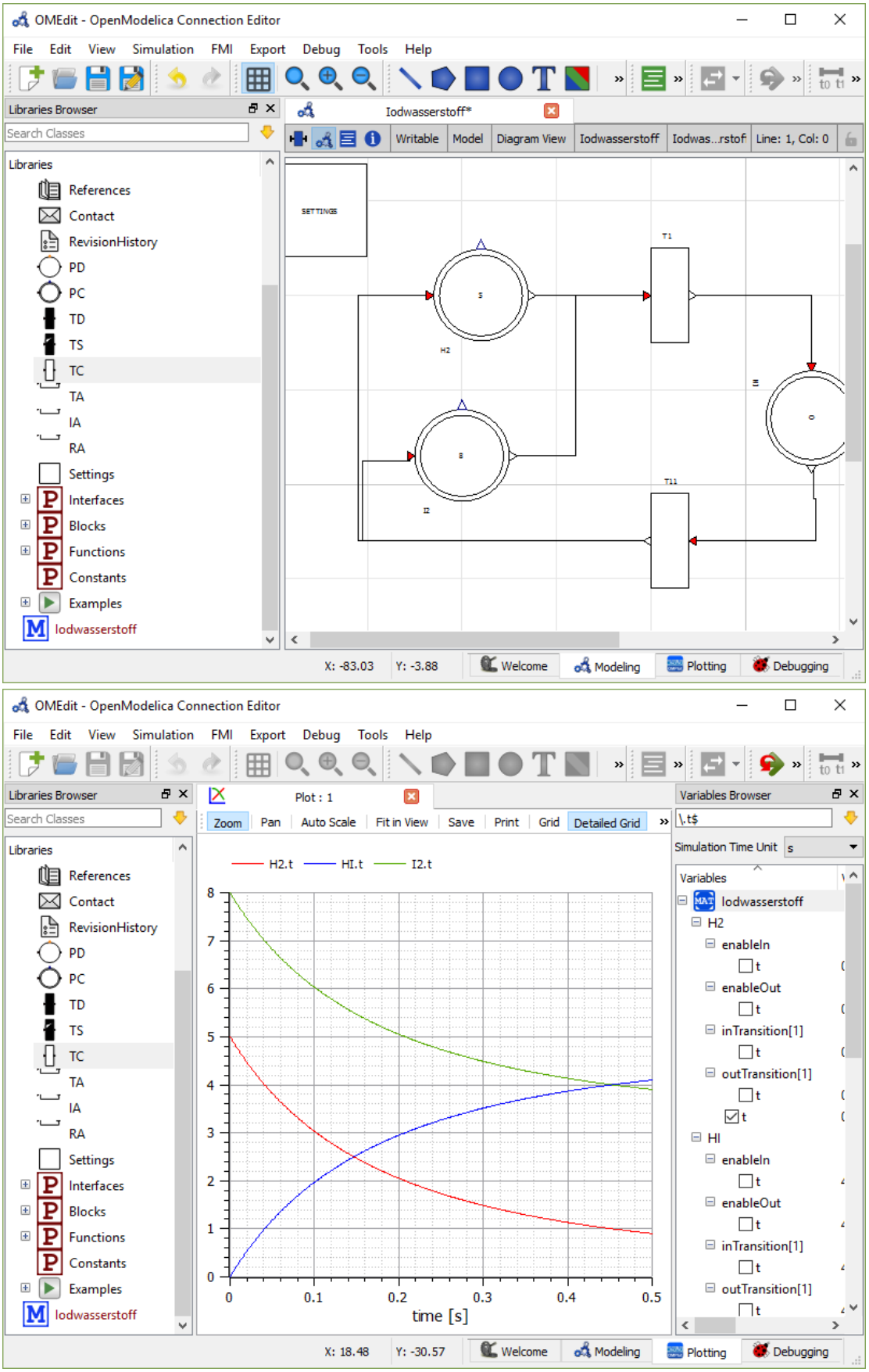

Abbildung 2.5: OMEdit mit einem Petri-Netz Modell. Oben ist das Blockdiagramm des Modells zu sehen und darunter die zugehörigen Simulationsergebnisse.

2.1.3.3 OMNotebook

OMNotebook [9, pp. 71-92] ist das elektronische Notizbuch von OpenModelica, mit dem interaktive Dokumentationen erstellt werden können.

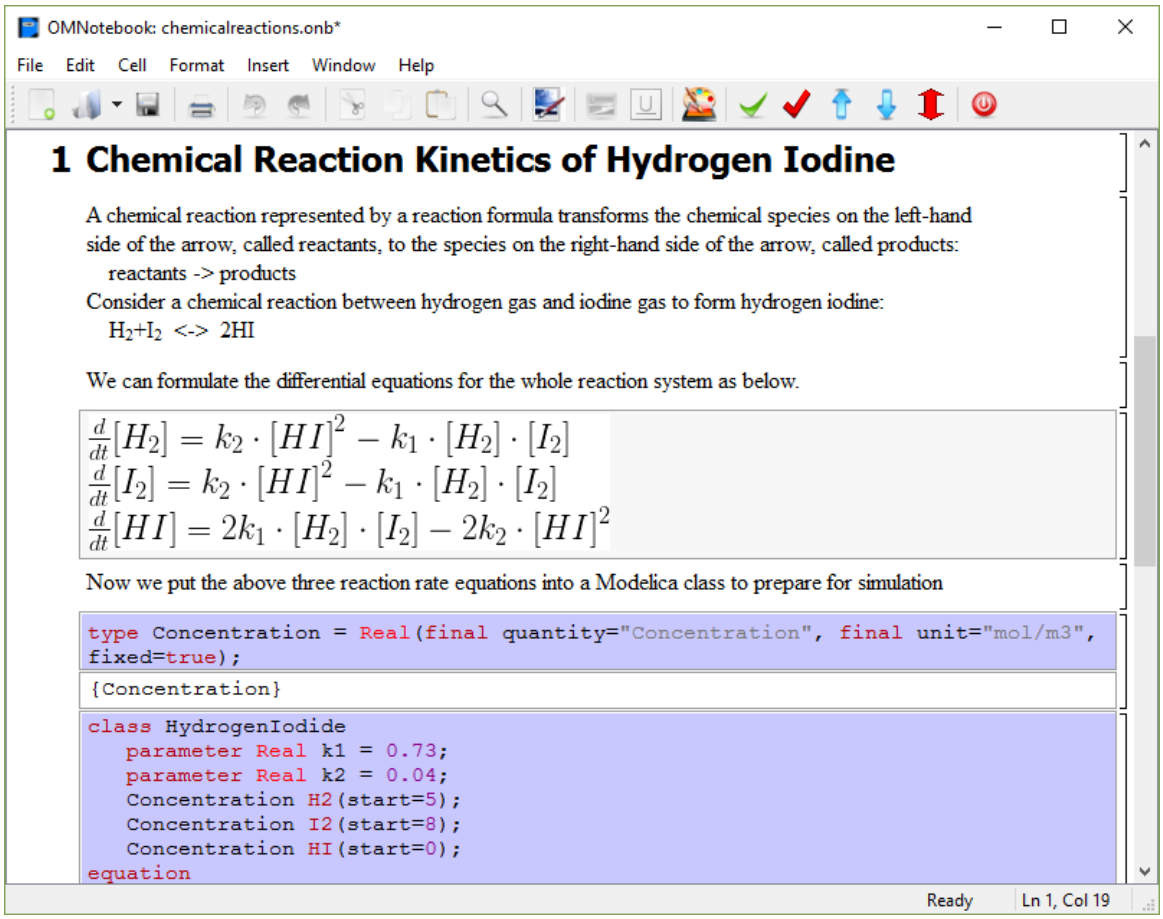

Abbildung 2.6: OMNotebook ist eine der grafischen Oberflächen von OpenModelica. Hier ist ein Notizbuch dargestellt, das die Möglichkeiten der Dokumentation mittels formatiertem Text, Latex-Elementen zur Darstellung von Gleichungen und interaktiv interpretiertem Modelica-Code nutzt, um ein Beispiel zur Iodwasserstoff-Produktion darzustellen.

Die Notizbücher bestehen aus einer hierarchischen Struktur von Zellen, die entweder Texte, Bilder, Latex-Inhalte oder auch interaktiv ausführbaren Modelica-Code enthalten können. Somit eignen sich OMNotebook-Notizbücher insbesondere zur Dokumentation und Weiterbildung.

2.2 Petri-Netze

Petri-Netze sind ein Formalismus zur Beschreibung von Systemzuständen und –verhalten, der ursprünglich von Carl Adam Petri [15] entwickelt wurde. Als besondere Stärke gilt die Beschreibung nebenläufiger Prozesse. Darüber hinaus finden Petri-Netze inzwischen durch eine Vielfalt von Modifikationen und Erweiterungen Einsatz in unterschiedlichsten Disziplinen, z.B. diskrete Petri-Netze zur Konstruktion asynchroner Hardware [16, p. 209ff], xHPN-Formalismus zur Pflegepersonalplanung [17] und gefärbte Petri-Netze zur Unterstützung der Ablaufkoordinierung der Australischen Verteidigungsstreitkraft [18, p. 350ff].

Ein Petri-Netz ist ein schwach zusammenhängender Graph, dessen Knoten sich in die beiden disjunkten Mengen der Plätze und Transitionen aufteilt. Die Knoten sind durch gerichtete Kanten verbunden, wobei jede Kante entweder einen Platz mit einer Transition oder eine Transition mit einem Platz verbindet. Für gewöhnlich werden Plätze als Kreise und Transitionen als Rechtecke dargestellt (siehe Abbildung 2.7). Jedem Platz kann eine bestimmte nichtnegative ganze Anzahl von Marken zugewiesen werden, wodurch der Zustand des Petri-Netzes definiert wird. Dieser Zustand kann sich durch das sogenannte Feuern der Transitionen ändern. Dies wird anhand des folgenden Beispiels anschaulich gemacht.

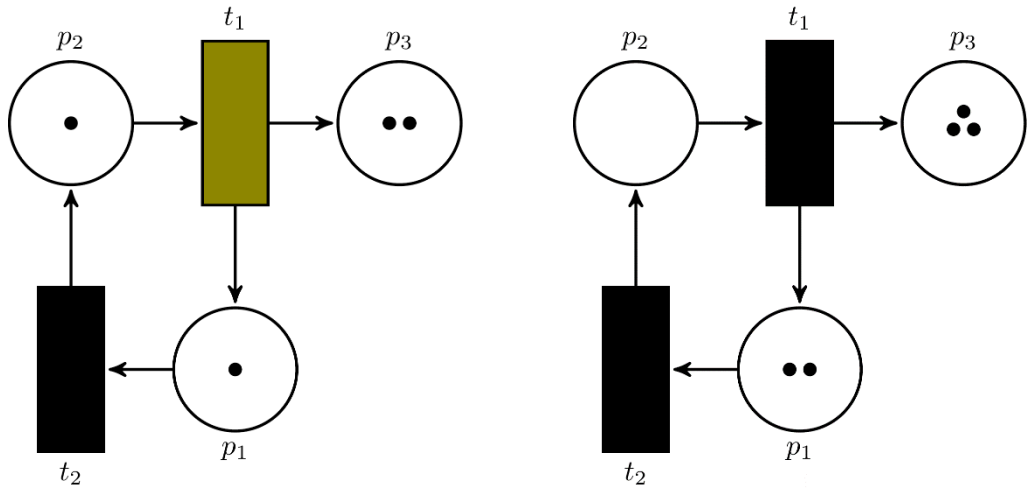

Abbildung 2.7: Beispiel eines (diskreten) Petri-Netzes mit drei Plätzen und zwei Transitionen. Links ist der Ausgangszustand des Netzes dargestellt und rechts der Zustand, nachdem die Transition t_1 gefeuert wurde. In dem rechten Bild könnte lediglich Transition t_2 feuern, da der Vorbereich von t_1 (also p_2) keine Marke mehr enthält.

Der Zustand des Petri-Netzes in Abbildung 2.7 wird durch die Anzahl der Marken in den Plätzen p_1, p_2 und p_3 beschrieben. Dieser Zustand ändert sich durch das Feuern einer oder beider Transitionen. Eine Transition kann immer dann feuern, wenn alle Plätze ihres Vorbereichs wenigstens eine Marken enthalten. Der Vorbereich beschreibt die Plätze, die mit einer gerichteten Kante in Richtung der jeweiligen Transition verbunden sind, wohingegen der Nachbereich die Plätze beschreibt, die mit einer gerichteten Kante ausgehend von der jeweiligen Transition verbunden sind. Beim Feuern einer Transition wird aus allen Plätzen des Vorbereichs je eine Marke entnommen und den Plätzen des Nachbereich wird jeweils eine Marke hinzugefügt.

Im Folgenden werden Petri-Netze, gemeinsam mit einigen wichtigen Erweiterungen, wie z.B. das Zeitkonzept und funktionale Kantengewichte, formal eingeführt.

2.2.1 Diskrete Petri-Netze

Die nachfolgende Einführung diskreter Petri-Netze orientiert sich an [19] und die Einführung zeitbasierter Petri-Netze, funktionaler Petri-Netze und kontinuierlicher Petri-Netze orientiert sich an [1]. Es wird eine einheitliche Notation verwendet, die an [19] angelehnt ist.

Definition 2.1: Netz (siehe [19, p. 5])

Ein **Netz** N ist ein Tupel $N = (P, T, F, B)$ mit einer endlichen Menge $P = \{p_1, \dots, p_m\}$ und einer endlichen Menge $T = \{t_1, \dots t_n\}$, wobei $P \cap T = \emptyset$; zudem gelte $F \subseteq P \times T$, $B \subseteq T \times P$.
Die Elemente aus P werden **Plätze** und die Elemente aus T werden **Transitionen** genannt. Die Elemente aus $F \cup B$ heißen Pfeile oder gerichtete Kanten von N.

Das Tupel $G = (V, E)$ mit $V = P \cup T$ und $E = F \cup B$ bildet einen 2-gefärbten Digraph.

Definition 2.2: Diskretes Petri-Netz (siehe [19, p. 6])

Ein **diskretes Petri-Netz** N ist ein Tupel $N = (P, T, F, B, f)$, wobei $N = (P, T, F, B)$ ein Netz gemäß Definition 2.1 ist. Die Abbildung $f: F \cup B \to \mathbb{N}$ heißt **Gewichtungsfunktion** von N, die jedem Pfeil $(p, t) \in P \times T$ bzw. $(t, p) \in T \times P$ eine natürliche Zahl als das zugehörige **Gewicht** $f(p, t)$ bzw. $f(t, p)$ zuweist.[1]

Der entsprechende gewichtete Digraph $G = (V, E, f)$ lässt sich als ungewichteter Multidigraph G^* interpretieren, indem jeder Pfeil $(i, j) \in E$ durch $f(i, j)$ parallele und ungewichtete Pfeile ersetzt wird. Entsprechend wird auch von der gewichteten bzw. ungewichteten Darstellungsform eines Petri-Netzes gesprochen.

Definition 2.3: Vor- und Nachbereich, Vor- und Nachbedingung (siehe [19, p. 6])

Es sei ein Petri-Netz $N = (P, T, F, B, f)$ gegeben.

a) Jedem Platz $p \in P$ wird durch $^\circ p = \{t \in T \mid (t, p) \in B\}$ ein sogenannter **Vorbereich** und durch $p^\circ = \{t \in T \mid (p, t) \in F\}$ ein sogenannter **Nachbereich** zugewiesen.
b) Jeder Transition $t \in T$ wird durch $^\circ t = \{p \in P \mid (p, t) \in F\}$ ein sogenannter **Vorbereich** und durch $t^\circ = \{p \in P \mid (t, p) \in B\}$ ein sogenannter **Nachbereich** zugewiesen.
c) Die Elemente aus einem Vor- bzw. Nachbereich werden auch **Vor-** bzw. **Nachbedingung** genannt.

Die folgende Definition führt Knotengewichte für die Plätze ein, welche den sogenannten Zustand eines Petri-Netzes beschreiben:

Definition 2.4: Zustand (siehe [19, p. 6])

Es sei ein Petri-Netz $N = (P, T, F, B, f)$ gegeben. Eine Abbildung $\boldsymbol{z}: P \to \mathbb{N}_0$ heißt **Zustand** oder **Markierung** von N, wobei $\boldsymbol{z}$ eindeutig als Vektor $\boldsymbol{z} = (z_1, \dots, z_m)$ mit $z_i = \boldsymbol{z}(p_i)$ darstellbar ist. Die Menge Z_r aller Abbildungen von P in $\mathbb{N}_0$ wird der **Zustandsraum** von N genannt.

[1] Es wird die vereinfachte Schreibweise $f(p, t)$ für $f\big((p, t)\big)$ beziehungsweise $f(t, p)$ für $f\big((t, p)\big)$ vereinbart.

Eine Markierung von N lässt sich als eine Knotenbewertung der Knotenmenge P auffassen. Die Knotenwerte $\boldsymbol{z}(p)$ spiegeln sich in der graphischen Darstellung des Petri-Netzes N als die Anzahl der Marken an den Plätzen $p \in P$ wieder. Die Marken werden häufig auch als Tokens bezeichnet.

Definition 2.5: Serielles Feuern (siehe [19, p. 8])

Es sei $N = (P, T, F, B, f)$ ein Petri-Netz und $\boldsymbol{z}$ ein Zustand von N.

a) Eine Transition $t \in T$ von N heißt **aktiviert** oder **seriell feuerbar** im Zustand $\boldsymbol{z}$, wenn für alle $p \in {}^\circ t$ gilt: $f(p,t) \leq \boldsymbol{z}(p)$.

b) Eine im Zustand $\boldsymbol{z}$ aktivierte Transition t wird kurz auch **$\boldsymbol{z}$-feuerbar** genannt.

c) Das **Feuern** einer $\boldsymbol{z}$-feuerbaren Transition $t \in T$ von N ist der Übergang vom Zustand $\boldsymbol{z} = (z_1, \dots, z_m)^\top$ in den Zustand $\boldsymbol{z}' = (z_1', \dots, z_m')^\top$, wobei für $i = 1, \dots, m$ gilt:

$$z_i' = \begin{cases} z_i - f(p_i,t) + f(t,p_i), & p_i \in {}^\circ t \wedge p_i \in t^\circ \\ z_i - f(p_i,t), & p_i \in {}^\circ t \wedge p_i \notin t^\circ \\ z_i + f(t,p_i), & p_i \notin {}^\circ t \wedge p_i \in t^\circ \\ z_i, & p_i \notin {}^\circ t \wedge p_i \notin t^\circ \end{cases}$$

Neben dem seriellen Feuern, bei dem immer nur genau eine aktivierte Transition bzgl. eines Zustandes $\boldsymbol{z}$ für einen Zustandsübergang herangezogen wird, soll nun auch nebenläufiges Feuern von mehreren Transitionen formalisiert werden. Hierzu wird der Begriff der Feuerbarkeit mit Definition 2.6 verallgemeinert.

Definition 2.6: Nebenläufiges Feuern (vgl. [19, p. 10])

Es sei $N = (P, T, F, B, f)$ ein Petri-Netz im Zustand $\boldsymbol{z}$ und $T_n \subseteq T$ eine Menge von Transitionen. Mit

$${}^\circ T_n = \bigcup_{t \in T_n} {}^\circ t = \{p \in P | \exists t \in T_n : (p,t) \in F\}$$

werde der **gemeinsame Vorbereich** der Transitionen T_n bezeichnet. Die Transitionsmenge T_n heißt (**nebenläufig**) **feuerbar** im Zustand $\boldsymbol{z}$, wenn alle Transitionen aus T_n seriell feuerbar sind und für alle $p \in {}^\circ T_n$ gilt:

$$\sum_{t \in T_n \cap p^\circ} f(p,t) \leq \boldsymbol{z}(p).$$

2.2.2 Petri-Netze mit Konfliktlösungen

Durch das Einführen der nebenläufigen Feuerbarkeit kann es zu sogenannten Konfliktsituationen (siehe [1], [19]) kommen. Dies ist der Fall, sobald eine Menge von aktiven Transitionen nebenläufig gefeuert werden soll, die Aufgrund ihrer indirekten Abhängigkeiten im Netzwerk dennoch nicht nebenläufig gefeuert werden können. Dies wird mit folgenden Beispiel verdeutlicht:

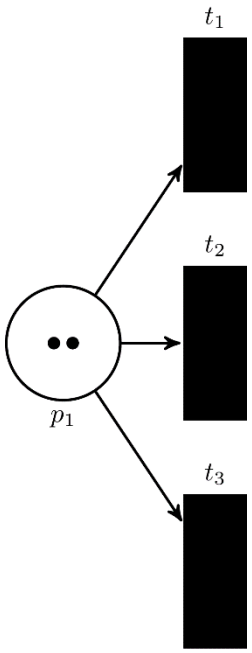

Abbildung 2.8: Diskretes Petri-Netz mit einem Konflikt, sofern alle drei Transitionen nebenläufig gefeuert werden sollen.

Das Beispiel aus Abbildung 2.8 zeigt, dass lediglich eine Teilmenge der nebenläufig zu feuernden Transitionen, dies sind in diesem Fall alle Transitionen, gefeuert werden kann. Wie diese Menge im konkreten Fall bestimmt wird, beschreibt die Konfliktlösung:

Definition 2.7: (diskretes) Petri-Netz mit Konfliktlösung (vgl. [19, p. 40f])

Das Tupel $(P, T, F, B, f, \boldsymbol{z}, \alpha, \delta, \zeta)$ ist ein **Petri-Netz mit Konfliktlösung**, wenn gilt:

a) (P, T, F, B, f) ist ein Petri-Netz im Zustand $\boldsymbol{z}$**.**

b) Die Abbildung $\alpha: P \to \{\text{Priorität}, \text{Wahrscheinlichkeit}\}$ ist eine **Bewertungsartfunktion**, welche jedem Platz $p \in P$ eine bestimmte Art von Bewertung zuordnet.

c) Die Abbildung $\delta: F \to \begin{cases} \mathbb{N}, & \alpha(p) = \text{Priorität} \\ [0,1] \subset \mathbb{R}, & \alpha(p) = \text{Wahrscheinlichkeit} \end{cases}$ ist eine **Bewertungsfunktion**, welche jedem Pfeil von einem Platz $p \in P$ zu einer Transition $t \in p^\circ$ eine Bewertung $\delta(p,t)$ entsprechend der Bewertungsart des Platzes und unter den folgenden Bedingungen zuordnet:

   a. Jede Priorität darf nur einmal von jedem Platz p benutzt werden:
   $$\delta(p, t_i) \neq \delta(p, t_j)\ \forall t_i, t_j \in p^\circ \text{ mit } t_i \neq t_j \text{ falls } \alpha(p) = \text{Priorität}.$$

   b. Die Summe der Wahrscheinlichkeiten von einem Platz p zu den Transitionen $t \in p^\circ$ muss gleich 1 sein:
   $$\sum_{t \in p^\circ} \sigma(p,t) = 1\ \forall p \text{ falls } \alpha(p) = \text{Wahrscheinlichkeit}.$$

Konkrete Algorithmen zur Behandlung dieser Konflikte können z.B. in [1] und [19] nachgeschlagen werden.

2.2.3 Zeitbasierte Petri-Netze

Bis hierhin wurde lediglich definiert, wie sich der Zustand beim Feuern ändert, nicht jedoch wann eine Transition tatsächlich feuert. Hierfür wird im Folgenden der Zeitbegriff eingeführt.

Definition 2.8: Zeitbasiertes Petri-Netz (vgl. [1, p. 100])

Das Tupel $(P, T, F, B, f, \boldsymbol{z}, \alpha, \delta, \zeta, d, time)$ ist ein **zeitbasiertes Petri-Netz**, wenn gilt:

a) $(P, T, F, B, f, \boldsymbol{z}, \alpha, \delta, \zeta)$ ist ein Petri-Netz mit Konfliktlösung gemäß Definition 2.7.
b) Die Abbildung $d: T \to \mathbb{R}^+$ ist eine **Verzögerungsfunktion**, die jeder Transition $t \in T$ eine nicht-negative Zahl zuordnet, die die Verzögerung der jeweiligen Transition beschreibt.
c) $\text{time} \coloneqq [a, b] \subseteq \mathbb{R}$ ist die Menge aller Zeitpunkte, kurz Zeit genannt. a ist der initiale Zeitpunkt und b der finale Zeitpunkt.

Definition 2.8 führt die Zeit global für das gesamte Petri-Netz ein. Dieses Zeitkonzept ist grundverschieden zu der diskreten Zeit aus den gefärbten Petri-Netzen von Kurt Jensen [18], bei denen jedes Token einen „Zeitstempel“ erhält, der durch das Feuern einer Transition manipuliert werden kann.

2.2.4 Funktionale Petri-Netze

Definition 2.9: Funktionales Petri-Netz (vgl. [1, p. 97])

Das Tupel $N = (P, T, F, B, f, \boldsymbol{z}, \alpha, \delta, \zeta, d, time)$ ist ein **funktionales Petri-Netz**, wenn gilt:

a) $(P, T, F, B, f, \boldsymbol{z}, \alpha, \delta, \zeta, d, time)$ ist ein zeitbasiertes Petri-Netz mit Konfliktlösung gemäß Definition 2.8.
b) Die Gewichtungsfunktion f ist dahingehend modifiziert, dass $f: (F \cup B, Z_r, time) \to \mathbb{N}_0$ eine **funktionale Gewichtungsfunktion** ist, die zusätzlich von dem Zustandsraum von N und der Zeit abhängt.

Es wird explizit die 0 als Kantengewicht zugelassen. Dies entspricht einer nichtvorhandenen Verbindung zwischen Platz und Transition in der ungewichteten Darstellungsform. Somit kann fortan auch kein statischer Vor- bzw. Nachbereich für ein Petri-Netz mehr angegeben werden, da dieser zusätzlich von dem Zustandsraum des Petri-Netzes und der Zeit abhängt.

2.2.5 Kontinuierliche Petri-Netze

Nun wird die Definition der Markierung auf den kontinuierlichen Fall übertragen:

Definition 2.10: Kontinuierlicher Zustand (vgl. Definition 2.4)

Es sei ein Petri-Netz $N = (P, T, F, B, f)$ gegeben. Eine Abbildung $\boldsymbol{z}: P \to \mathbb{R}_0$ heißt (kontinuierlicher) **Zustand** oder **Markierung** von N, wobei $\boldsymbol{z}$ eindeutig als Vektor $\boldsymbol{z} = (z_1, \dots, z_m)$ mit $z_i = \boldsymbol{z}(p_i)$ darstellbar ist. Die Menge Z_r aller Abbildungen von P in $\mathbb{R}_0$ wird der **Zustandsraum** von N genannt.

In kontinuierlichen Petri-Netzen ist nicht nur der Zustand kontinuierlich, sondern auch die Transitionen. Dies bedeutet, sie feuern nicht mit einer definierten Verzögerung, sondern

innerhalb eines definierten Zeitfensters. Somit ergibt sich aus dem Übergang des diskreten Petri-Netz-Konzeptes, dass Transitionen eine Geschwindigkeit anstatt einer Verzögerung zugewiesen bekommen. Dies resultiert direkt aus der Überlegung immer kleiner werdender Verzögerungen, bis hin zu kontinuierlichen Transitionen (vgl. [1, p. 106ff]).

Definition 2.11: Kontinuierliches Petri-Netz (vgl. [1, p. 107])

Das Tupel $(P, T, F, B, f, v, \boldsymbol{z}, time)$ ist ein **kontinuierliches Petri-Netz**, wenn gilt:

a) Das Tupel (P, T, F, B) ist ein Netz gemäß Definition 2.1.
b) $f: (F \cup B, Z_r, time) \to \mathbb{R}_0^+$ ist **funktionale Gewichtungsfunktion**
c) $\boldsymbol{z}$ ist der Zustand des Petri-Netzes.
d) $v: T \to \mathbb{R}_0^+$ ist die **maximale Geschwindigkeitsfunktion**, die jeder Transition $t \in T$ die maximale Geschwindigkeit $v(t)$ zuordnet.

2.2.6 Der xHPN-Formalismus

Die oben beschriebenen Erweiterungen wurden in [1] zusammen mit weiteren Konzepten wie z.B. Plätze mit Kapazitäten, stochastische Transitionen und Konfliktlösungen, zu dem xHPN-Formalismus zusammengefasst. Somit ist der xHPN-Formalismus deutlich mächtiger, als die hier eingeführten Petri-Netze. Allerdings lassen sich die im weiteren Verlauf dieser Arbeit eingeführten Konzepte problemlos auf den xHPN-Formalismus übertragen. Für eine vollständige Beschreibung der obigen Konzepte bezogen auf den xHPN-Formalismus sei hier auf die Definitionen aus [1] verwiesen:

- Konfliktlösung in diskreten Petri-Netzen, siehe Def. 4.11 (resolved Petri net)
- Konfliktlösung in xHPN, siehe Def. 4.11 (resolved Petri net)
- Funktionale Petri-Netze, siehe Def. 4.28 (functional Petri net)
- Zeitbasierte Petri-Netze, siehe Def. 4.32 (timed Petri net)
- Kontinuierliche Petri-Netze, siehe Def. 4.38 (continous Petri net)

2.3 Mathematische Modellierung von Reaktionskinetiken

Deterministische Modelle einzelner biochemischer Reaktionen werden bereits seit langer Zeit formuliert. So ist beispielsweise das Michaelis-Menten-Modell für die Reaktionsrate einer irreversiblen Ein-Substrat-Reaktion ein zentraler Bestandteil der heutigen Biochemie, und der sogenannte K_m-Wert ist ein wesentliches Charakteristikum für die wechselseitige Beziehung zwischen Enzym und Substrat. Enzyme sind spezielle Proteine, die biochemische Reaktionen katalysieren. Ein einzelnes Enzym-Molekül kann tausende Reaktionen pro Sekunde katalysieren. Diese Eigenschaft wird als Wechselzahl (engl. turnover number) bezeichnet und reicht von $10^2 s^{-1}$ bis $10^7 s^{-1}$. Somit beschleunigen Enzyme eine Reaktion, verglichen mit der nicht-katalysierten spontanen Reaktion, um das 10^6 bis 10^{12}-fache. (vgl. [20, p. 13])

Im Folgenden werden grundlegende mathematische Zusammenhänge einfacher Reaktionskinetiken beschrieben. Zunächst wird auf das Massenwirkungsgesetz eingegangen und anschließend darauf aufbauend die Michaelis-Menten-Gleichungen hergeleitet. An zwei Beispielen wird gezeigt, wie sich Reaktionen auf Basis dieser Kinetiken durch unterschiedliche Modellierungsansätze abbilden lassen.

2.3.1 Massenwirkungsgesetz

Biochemische Reaktionskinetiken basieren auf dem Massenwirkungsgesetz, welches von Gulberg und Waage in dem 19. Jahrhundert formuliert wurde [21, p. 194]. Es besagt, dass die Reaktionsrate proportional zu der Kollisionswahrscheinlichkeit der Reaktionspartner, den sogenannten Edukten, ist. Diese Wahrscheinlichkeit wiederum, ist proportional zu der Konzentration der Edukte potenziert mit dem entsprechenden stöchiometrischen Koeffizienten.

$$A + B \underset{k_{\leftarrow}}{\overset{k_{\rightarrow}}{\rightleftharpoons}} 2C \tag{2.1}$$

Somit ergibt sich für die einfache Gleichgewichtsreaktion aus Gleichung (2.1), die nachfolgenden Reaktionsraten:

$$v_{\rightarrow} = k_{\rightarrow}[A][B] \tag{2.2}$$
$$v_{\leftarrow} = k_{\leftarrow}[C]^2 \tag{2.3}$$

Hierbei ist $v_{\rightarrow}$ die Reaktionsrate der Hinreaktion und $v_{\leftarrow}$ die Reaktionsrate der Rückreaktion. $[A]$, $[B]$ und $[C]$ sind die Stoffkonzentrationen und $k_{\rightarrow}$ und $k_{\leftarrow}$ die zugehörigen Proportionalitätsfaktoren. A und B sind unimolekular an der Reaktion beteiligt. Daher gehen diese Stoffkonzentrationen jeweils mit dem Exponenten 1 in die Reaktionsrate ein. Dahingegen kommt C bimolekular vor und geht daher quadratisch ein.

$$v = v_{\rightarrow} - v_{\leftarrow} \tag{2.4}$$

Die effektive Reaktionsrate v ergibt sich somit wie in Gleichung (2.4) gezeigt. Allgemein sieht das Massenwirkungsgesetz wie folgt aus:

$$v = v_{\rightarrow} - v_{\leftarrow} = k_{\rightarrow} \prod_{e \in E} [e]^{n_e} - k_{\leftarrow} \prod_{p \in P} [p]^{n_p} \tag{2.5}$$

Hierbei sind E und P Mengen aller involvierter Edukte bzw. Produkte, n_e und n_p die Molekularität und $[e]$ und $[p]$ die Konzentration des Eduktes e bzw. des Produktes p.

Beispiel Iodwasserstoff

Die Modellierung des zeitabhängigen Verhaltens einer Reaktion auf Basis des Massenwirkungsgesetzes wird anhand einer einfachen chemischen Reaktionsgleichung zur Gewinnung von Iodwasserstoff demonstriert. Das Beispiel ist [6, pp. 829-831] entnommen:

$$\mathrm{H_2 + I_2} \underset{k_2}{\overset{k_1}{\rightleftharpoons}} \mathrm{2HI} \tag{2.6}$$

Der beidseitige Reaktionspfeil ist eine Kurzschreibweise für die folgenden zwei Reaktionen:

$$\mathrm{H_2 + I_2} \overset{k_1}{\rightarrow} \mathrm{2HI} \tag{2.7}$$

$$\mathrm{2HI} \overset{k_2}{\rightarrow} \mathrm{H_2 + I_2} \tag{2.8}$$

Die Hinreaktion (2.7) beschreibt, dass ein Wasserstoff-Molekül mit einem Iod-Molekül reagieren kann. Dabei gehen die einzelnen Moleküle eine neue Verbindung ein und so entstehen aus ihnen zwei Iodwasserstoff-Moleküle. Die zweite Reaktion (2.8) ist die Rückreaktion, bei der zwei Moleküle Iodwasserstoff zu molekularem Wasserstoff und Iod zerfällt.

Mit Hilfe dieser Reaktionsgleichungen und dem Massenwirkungsgesetz aus Gleichung (2.5) lässt sich das folgende System von gewöhnlichen Differentialgleichungen, zur Beschreibung des dynamischen Verhaltens der Stoffkonzentrationen, formulieren:

$$\frac{d[\mathrm{H_2}]}{dt} = k_2 \cdot [\mathrm{HI}]^2 - k_1 \cdot [\mathrm{H_2}] \cdot [\mathrm{I_2}] \tag{2.9}$$

$$\frac{d[\mathrm{I_2}]}{dt} = k_2 \cdot [\mathrm{HI}]^2 - k_1 \cdot [\mathrm{H_2}] \cdot [\mathrm{I_2}] \tag{2.10}$$

$$\frac{d[\mathrm{HI}]}{dt} = 2k_1 \cdot [\mathrm{H_2}] \cdot [\mathrm{I_2}] - 2k_2 \cdot [\mathrm{HI}]^2 \tag{2.11}$$

Gleichung (2.9) beschreibt die Änderung der $\mathrm{H_2}$-Konzentration. Diese setzt sich aus zwei Teilen zusammen: Produktionsrate und Abbaurate. Die Produktionsrate ergibt sich aus Gleichung (2.7) und lautet $k_2 \cdot [\mathrm{HI}]^2$. Die Abbaurate wird durch Gleichung (2.8) beschrieben und lautet $k_1 \cdot [\mathrm{H_2}] \cdot [\mathrm{I_2}]$. Gleichung (2.10) ist analog dazu aufgebaut und beschreibt die I_2-Konzentrationsänderung.

Gleichung (2.11) beschreibt die Änderung der HI-Konzentration und setzt sich ebenfalls aus einer Produktionsrate und einer Abbaurate zusammen. Der Faktor 2 taucht in dem Produktionsterm auf, da jeweils zwei HI-Moleküle aus einem $\mathrm{H_2}$ und einem $\mathrm{I_2}$ Molekül gebildet werden. Analog begründet sich der Faktor 2 im zweiten Term.

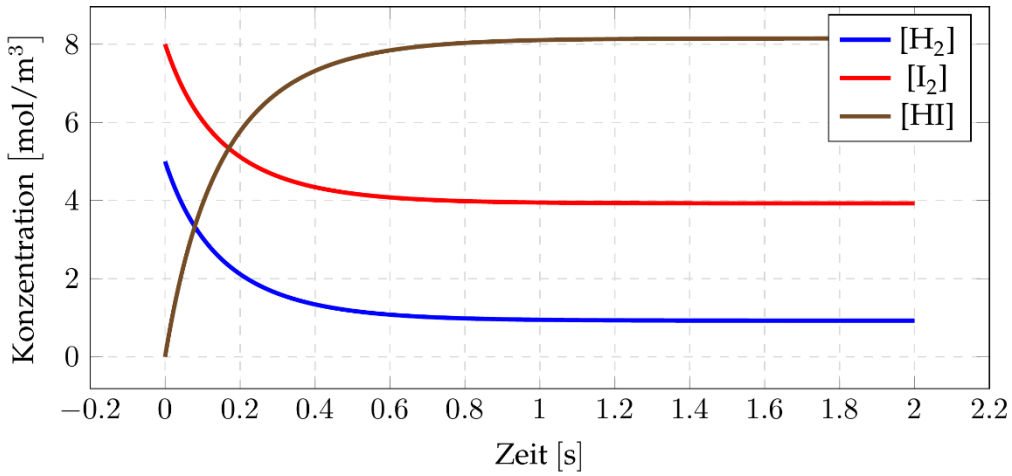

Abbildung 2.9: Konzentrationsverläufe des Beispiels Iodwasserstoff mit den Anfangsbedingungen aus (2.12)-(2.14). Nach kurzer Zeit stellt sich ein Konzentrationsgleichgewicht ein.

Das mathematische Modell aus den Gleichungen (2.9)-(2.11) beschreibt das dynamische Systemverhalten. Unter Hinzunahme von konkreten Anfangswerten der einzelnen Stoffkonzentrationen kann das System für ein definiertes Zeitfenster berechnet werden. Konkret ergeben sich für das Zeitfenster $[0; 2]$ und den Startwerten aus Gleichungen (2.12)-(2.14) die Konzentrationsverläufe aus Abbildung 2.9.

$$[\mathrm{H}_2](t_0) = 5 \quad (2.12)$$
$$[\mathrm{I}_2](t_0) = 8 \quad (2.13)$$
$$[\mathrm{HI}](t_0) = 0 \quad (2.14)$$

Da Modelica gleichungsbasiert ist und gewöhnliche Differentialgleichungen unterstützt, lässt sich dieses mathematische Modell leicht in ein Modelica-Modell überführen, wie Listing 1 verdeutlicht.

```
1  type Concentration = Real(final quantity="Concentration",
2    final unit="mol/m3", fixed=true);
3
4  model Hydrogen_Iodide
5    parameter Real k1 = 0.73;
6    parameter Real k2 = 0.04;
7    Concentration H2(start=5);
8    Concentration I2(start=8);
9    Concentration HI(start=0);
10 equation
11   der(H2) = k2*HI^2 - k1*H2*I2;
12   der(I2) = k2*HI^2 - k1*H2*I2;
13   der(HI) = 2*k1*H2*I2 - 2*k2*HI^2;
14 end Hydrogen_Iodide;
```

Listing 2.1: Das Beispiel Iodwasserstoff als Modelica-Modell.

Einfacher ist die Modellierung mit Hilfe von kontinuierlichen Petri-Netzen, wie nachstehende Grafik zeigt. Bei diesem Modellierungsansatz müssen keine Differentialgleichungen für die

einzelnen Stoffkonzentrationen aufgestellt werden, da dies automatisch durch den Petri-Netz-Formalismus geschieht. Es müssen lediglich die Stöchiometrie an die Kanten und die Reaktionsraten an die Transitionen geschrieben werden.

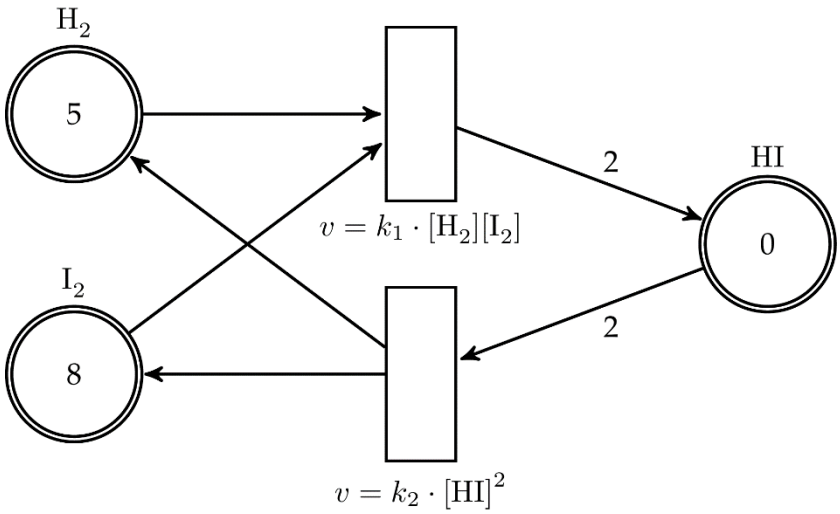

Abbildung 2.10: Das Beispiel Iodwasserstoff als kontinuierliches Petri-Netz.

2.3.2 Irreversible Michaelis-Menten-Gleichung

$$\mathrm{S} \overset{v_{max},\ K_m}{\rightarrow} \mathrm{P} \tag{2.15}$$

Die irreversible Michaelis-Menten-Gleichung (2.15) beschreibt den zeitlichen Verlauf einer durch ein Enzym E katalysierten Reaktion, die das Substrat S in das Produkt P überführt. Der Vorgang wird, wie in Gleichung (2.16) veranschaulicht, dadurch erklärt, dass das Enzym mit dem Substrat bindet und so einen Enzym-Substrat-Komplex ES bildet. In einem weiteren Schritt durchläuft dieser Komplex die eigentliche Transformation, in der aus dem Substrat das Produkt gebildet wird. Am Ende dieses Prozesses löst sich das Enzym wieder aus dem Komplex und gibt so das Produkt frei. Das Enzym wird in der Reaktion nicht verbraucht, sodass es für eine weitere Katalyse zur Verfügung steht. (vgl. [20, pp. 18-20])

$$\mathrm{E} + \mathrm{S} \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} ES \overset{k_2}{\rightarrow} \mathrm{E} + \mathrm{P} \tag{2.16}$$

Mit Hilfe des Massenwirkungsgesetztes kann aus dieser Reaktionskette das nachstehende System von Differentialgleichungen abgeleitet werden:

$$\frac{d[\mathrm{S}]}{dt} = k_{-1} \cdot [ES] - k_1 \cdot [S][E] \tag{2.17}$$

$$\frac{d[\mathrm{E}]}{dt} = k_{-1} \cdot [ES] + k_2 \cdot [ES] - k_1 \cdot [S][E] \tag{2.18}$$

$$\frac{d[\mathrm{ES}]}{dt} = k_1 \cdot [S][E] - k_{-1} \cdot [ES] - k_2 \cdot [ES] \tag{2.19}$$

$$\frac{d[\mathrm{P}]}{dt} = k_2 \cdot [ES] \tag{2.20}$$

Dem Michaelis-Menten-Ansatz liegt die Quasi-Steady-State-Annahme zugrunde, demnach sich die Enzymsubstratkonzentration schnell in einen stationären Zustand (siehe Gleichung (2.21)) einpendelt [22]. Diese Annahme gilt sofern die Substratkonzentrationen die Gesamtenzymkonzentration dominiert.

$$\frac{d[\mathrm{ES}]}{dt} = 0 \tag{2.21}$$

Somit lässt sich eine Gesamtreaktionsgeschwindigkeit v für den Produktaufbau, und somit gleichbedeutend mit dem Substratabbau, wie folgt beschreiben:

$$v = \frac{d[\mathrm{P}]}{dt} = -\frac{d[\mathrm{S}]}{dt} \tag{2.22}$$

Durch Einsetzen von Gleichung (2.21) in Gleichung (2.19) kann die Enzym-Substrat-Komplex-Konzentration wie folgt ausgedrückt werden:

$$[\mathrm{ES}] = \frac{k_1 \cdot [\mathrm{S}][\mathrm{E}]}{k_{-1} + k_2} \tag{2.23}$$

Unter Annahme einer konstanten Gesamtmenge an Enzym $[\mathrm{E}_t]$ lässt sich der folgende Zusammenhang zwischen der Enzymkonzentration $[\mathrm{E}]$ und der Konzentration des Enzym-Substrat-Komplexes $[\mathrm{ES}]$ ausdrücken:

$$[\mathrm{E}] = [\mathrm{E}_t] - [\mathrm{ES}] \tag{2.24}$$

Dies erlaubt das Eliminieren der Enzymkonzentration $[\mathrm{E}]$ aus Gleichung (2.23):

$$[\mathrm{ES}] = \frac{k_1 \cdot [\mathrm{S}][\mathrm{E}_t]}{k_{-1} + k_2 + k_1 \cdot [\mathrm{S}]} = \frac{[\mathrm{S}][\mathrm{E}_t]}{(k_{-1} + k_2)/k_1 + [\mathrm{S}]} \tag{2.25}$$

Somit lässt sich nun die Reaktionsgeschwindigkeit wie folgt berechnen:

$$v = \frac{k_2 \cdot [\mathrm{S}][\mathrm{E}_t]}{(k_{-1} + k_2)/k_1 + [\mathrm{S}]} \tag{2.26}$$

In dieser Form findet die Gleichung für die Reaktionsgeschwindigkeit allerdings kaum Anwendung, da im Allgemeinen die Reaktionskonstanten k_1, k_{-1}, k_2 und die Gesamtenzymkonzentration $[E_t]$ unbekannt sind.

$$\begin{aligned} v_{max} &= k_2[E_t] \\ K_m &= \frac{k_{-1} + k_2}{k_1} \end{aligned} \tag{2.27}$$

Die Konstanten v_{max} und K_m hingegen, lassen sich experimentell bestimmen und haben eine anschauliche Bedeutung in Bezug auf die Reaktionsgeschwindigkeit (siehe Abbildung 2.11 und vgl. [20, p. 20]), die somit wie folgt ausgedrückt werden kann:

$$v = \frac{v_{max} \cdot [\mathrm{S}]}{[\mathrm{S}] + K_m} \tag{2.28}$$

Die charakteristischen Parameter der Reaktionsgeschwindigkeit sind v_{max} und K_m. Die nachfolgende Grafik veranschaulicht ihre Bedeutung. v_{max} beschreibt die maximale Reaktionsgeschwindigkeit, gegen die v mit steigender Substratkonzentration strebt. K_m beschreibt die Substratkonzentration, bei der die Reaktionsgeschwindigkeit den halben Wert von v_{max} annimmt.

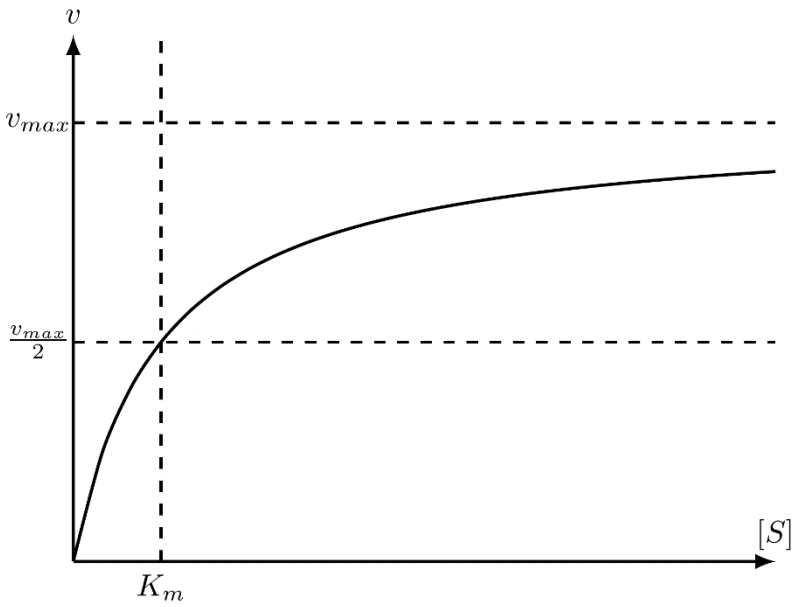

Abbildung 2.11: Grafische Veranschaulichung der charakteristischen Parameter v_{max} und K_m der irreversiblen Michaelis-Menten-Kinetik.

2.3.3 Reversible Michaelis-Menten-Gleichung

$$\mathrm{S} \underset{v^r_{max},\ K_{mP}}{\overset{v^f_{max},\ K_{mS}}{\rightleftharpoons}} \mathrm{P} \tag{2.29}$$

Der reversible Ansatz ist auf der gleichen Überlegung wie der irreversible Ansatz aufgebaut. Hinzu kommt lediglich die Rückreaktion, die aus dem Enzym E und dem Produkt P wieder den Enzym-Substrat-Komplex bildet:

$$\mathrm{E} + \mathrm{S} \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} ES \underset{k_{-2}}{\overset{k_2}{\rightleftharpoons}} \mathrm{E} + \mathrm{P} \tag{2.30}$$

Hierdurch müssen in den oben eingeführten Differentialgleichungen weitere Terme berücksichtigt werden:

$$\frac{d[\mathrm{S}]}{dt} = k_{-1} \cdot [ES] - k_1 \cdot [S][E] \tag{2.31}$$

$$\frac{d[\mathrm{E}]}{dt} = k_{-1} \cdot [ES] + k_2 \cdot [ES] - k_1 \cdot [S][E] - k_{-2} \cdot [E][P] \tag{2.32}$$

$$\frac{d[\mathrm{ES}]}{dt} = k_1 \cdot [S][E] + k_{-2} \cdot [P][E] - k_{-1} \cdot [ES] - k_2 \cdot [ES] \tag{2.33}$$

$$\frac{d[\mathrm{P}]}{dt} = k_2 \cdot [ES] - k_{-2} \cdot [P][E] \tag{2.34}$$

Bei dem reversiblen Ansatz wird, genauso wie beim irreversiblen Ansatz, ein Gleichgewicht des Enzym-Substrat-Komplexes angenommen. Somit kann $[\mathrm{ES}]$ wie nachstehend dargestellt werden:

$$[ES] = \frac{k_1 \cdot [S][E] + k_{-2} \cdot [P][E]}{k_2 + k_{-1}} \tag{2.35}$$

Die Menge freien Enzyms wird wieder mit Hilfe von Gleichung (2.24) substituiert:

$$[ES] = \frac{k_1 \cdot [S][E_t] + k_{-2} \cdot [P][E_t]}{k_2 + k_{-1} + k_1 \cdot [S] + k_{-2} \cdot [P]} \tag{2.36}$$

Die Geschwindigkeitsgleichung für den Produktaufbau lässt sich wie folgt aus Gleichung (2.34) herleiten:

$$\begin{aligned} v &= \frac{d[\mathrm{P}]}{dt} \\ &= k_2 \cdot [ES] - k_{-2} \cdot [P][E] \\ &= [ES] \cdot (k_2 + k_{-2} \cdot [P]) - k_{-2} \cdot [P][E_t] \end{aligned} \tag{2.37}$$

Durch das Einsetzen der Enzym-Substrat-Komplex-Konzentration $[\mathrm{ES}]$ aus Gleichung (2.36) in Gleichung (2.37) ergibt sich folgender Zusammenhang:

$$v = \frac{k_1 \cdot [S][E_t] + k_{-2} \cdot [P][E_t]}{k_2 + k_{-1} + k_1 \cdot [S] + k_{-2} \cdot [P]} \cdot (k_2 + k_{-2} \cdot [P]) \; - \; k_{-2} \cdot [P][E_t] \tag{2.38}$$

Durch algebraische Äquivalenzumformungen ergibt sich folgende Darstellung der Reaktionsgeschwindigkeit v:

$$v = \frac{[E_t] \cdot (k_1 \cdot \; k_2 \cdot \; [S] \; - \; k_{-1} \cdot \; k_{-2} \cdot \; [P])}{k_2 \; + \; k_{-1} + k_1 \cdot [S] + k_{-2} \cdot [P]} \tag{2.39}$$

Wie schon bei der irreversiblen Reaktion, findet diese Darstellung kaum Anwendung. Die in der Regel unbekannten Parameter $[\mathrm{E}_t]$, k_1, k_{-1}, k_2 und k_{-2} lassen sich allerdings durch folgende Größen darstellen, die in der Praxis relevant sind:

$$\begin{aligned} v^f &= [E_t] \cdot k_2 \\ v^r &= [E_t] \cdot k_{-1} \\ K_{mS} &= \frac{k_{-1} + k_2}{k_1} \\ K_{mP} &= \frac{k_{-1} + k_2}{k_{-2}} \end{aligned} \tag{2.40}$$

Somit ergibt sich durch Substitution dieser Terme die in der Literatur übliche Standardform (vgl. [20, p. 22]):

$$v = \frac{v^f_{max}/K_{mS} \cdot [S] \; - v^r_{max}/K_{mP} \cdot [P]}{1 + [S]/K_{mS} + [P]/K_{mP}} \tag{2.41}$$

Die Reaktionskinetik lässt sich für den Sonderfall, dass keine Produktkonzentration vorliegt, in die Reaktionskinetik der irreversiblen Reaktion überführen. Somit lassen sich auch die charakteristischen Parameter v^f_{max} und K_{mS} durch Abbildung 2.11 verdeutlichen. Gleiches gilt analog für die übrigen zwei Parameter, sofern keine Substratkonzentration vorliegt.

Beispiel Glucose-6-phosphat-Isomerase

Das Enzym Glucose-6-phosphat-Isomerase (PGI) katalysiert die Umwandlung von Glucose-6-phosphat (G6P) in Fructose-6-phosphat (F6P). Dieser Schritt ist Teil der Glykolyse, die Bestandteil des Stoffwechsels vieler Organismen ist und benötigt wird, um Energie aus Kohlenhydraten zu verwerten. Die Reaktion ist reversibel und wird nachfolgend mathematisch beschrieben:

$$\mathrm{G6P} \rightleftharpoons \mathrm{F6P} \tag{2.42}$$

Diese Enzymreaktion kann mit dem obigen Ansatz über die reversible Michaelis-Menten-Gleichung beschrieben werden:

$$\frac{d[\mathrm{G6P}]}{dt} = \frac{v_{max}^{r}/K_{mP} \cdot [\mathrm{G6P}] - v_{max}^{f}/K_{mS} \cdot [\mathrm{F6P}]}{1 + [G6P]/K_{mS} + [F6P]/K_{mP}} \tag{2.43}$$

$$\frac{d[\mathrm{F6P}]}{dt} = \frac{v_{max}^{f}/K_{mS} \cdot [\mathrm{F6P}] - v_{max}^{r}/K_{mP} \cdot [\mathrm{G6P}]}{1 + [G6P]/K_{mS} + [F6P]/K_{mP}} \tag{2.44}$$

Die konkreten Parameter für die nachstehenden Simulationsergebnisse sind [23] entnommen:

$$\begin{aligned} v_{max}^{f} &= 3815.708\ \mathrm{mM} \cdot \mathrm{min}^{-1} \\ v_{max}^{r} &= 496.042\ \mathrm{mM} \cdot \mathrm{min}^{-1} \\ K_{mS} &= 0.15\ \mathrm{mM} \\ K_{mP} &= 0.8\ \mathrm{mM} \end{aligned} \tag{2.45}$$

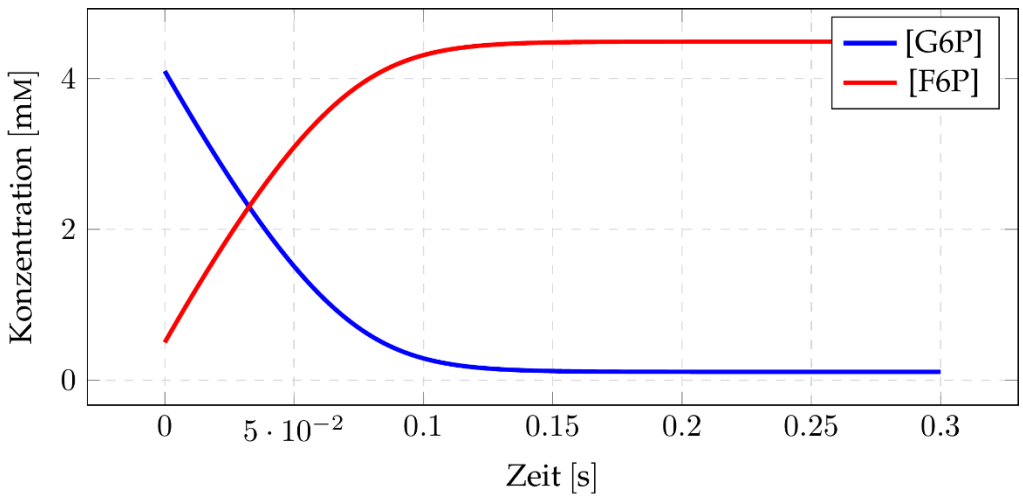

Abbildung 2.12: Konzentrationsverläufe zu dem Beispiel Glucose-6-phosphat-Isomerase.

Das Modelica-Modell ergibt sich direkt durch die Differentialgleichungen (2.43) und (2.44):

```
1  type Concentration = Real (
2    final quantity="Concentration",
3    final unit="mM", fixed=true);
4
5  model Isomerase
6    parameter Real v_f(final unit="mM/s") = 3815.708/60;
7    parameter Real v_r(final unit="mM/s") = 496.042/60;
8    parameter Real K_S(final unit="mM") = 0.15;
9    parameter Real K_P(final unit="mM") = 0.8;
10   Concentration G6P(start=4.1);
11   Concentration F6P(start=0.5);
12 equation
13   der(F6P) = (v_f/K_S*G6P - v_r/K_P*F6P) / (1 + G6P/K_S
14     + F6P/K_P);
15   der(G6P) = (v_r/K_P*F6P - v_f/K_S*G6P) / (1 + G6P/K_S
16     + F6P/K_P);
17 end Isomerase;
```

Listing 2.2: Beispiel Glucose-6-phosphat-Isomerase als Modelica-Modell.

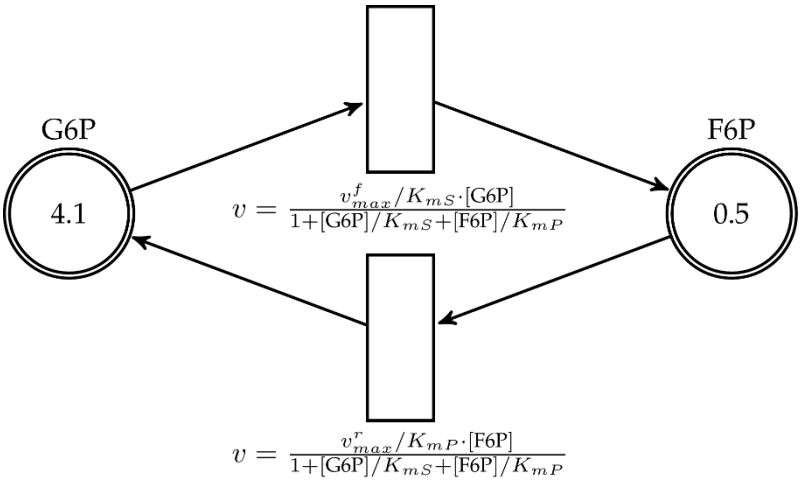

Abbildung 2.13: Beispiel Glucose-6-phosphat-Isomerase als kontinuierliches Petri-Netz.

In dem Petri-Netz-Modell werden Hin- und Rückreaktion getrennt voneinander modelliert.

3 Verwandte Arbeiten

Es gibt eine Reihe von Programmen, die sich mit unterschiedlichen Aspekten der Simulation von biologischen Systemen beschäftigen. In diesem Abschnitt wird auf solche Programme zur Modellierung und Simulation biologischer Netzwerke eingegangen. Das Hauptaugenmerk liegt hierbei auf Petri-Netz basierten Ansätzen.

Neben Programmen existieren noch Modellierungs-Bibliotheken, die sich zur Modellierung und Simulation von biologischen Netzwerken eignen. Bei den Bibliotheken steht nicht der Simulator, sondern das eigentliche Modell im Vordergrund. Die Bibliotheken bieten einen Formalismus, mit dem biologische Netze erstellt werden können, die einen definierten Simulationsverlauf beschreiben. Dieser Simulationsverlauf wird von einem entsprechenden Interpreter/Compiler berechnet, im Falle von Modelica-Bibliotheken z.B. durch den OpenModelica Compiler. Abhängig von der zugrundeliegenden Modellierungssprache, ist die Simulation nicht an ein bestimmtes Programm gebunden.

3.1 BioChem

BioChem [24], [8], [25] ist eine Modelica-Bibliothek zur Modellierung biologischer Prozesse auf Basis biologischer Netze. Hierfür stellt die Bibliothek die Pakete aus Abbildung 3.1 bereit, mit denen unter anderem Kompartimente, Reaktionen und Substanzen abgebildet werden können.

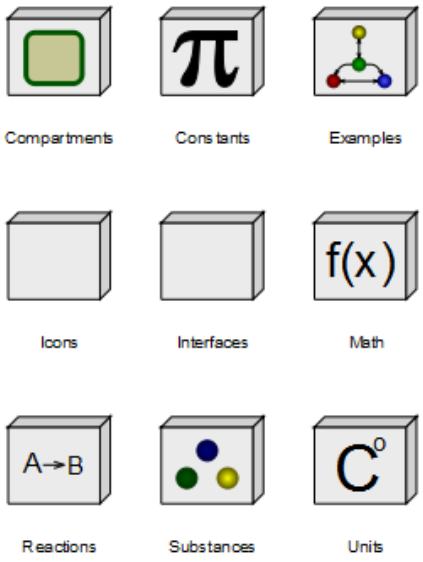

Abbildung 3.1: Aufbau der Modelica-Bibliothek BioChem.

Zum leichten Einstieg in die Modellierung mit Hilfe der BioChem-Bibliothek stellt das Paket `Examples` eine Reihe von Beispielen zur Verfügung, wie `BioChem.Examples.Yeast-Glycolysis.Cytosol`, das in Abbildung 3.2 zu sehen ist. Dieses Modell ist ein Teilmodell der Glykolyse und beinhaltet unter anderem die Glucose-6-phosphat-Isomerase, die bereits in den Grundlagen als Beispiel diente. Abbildung 3.3 zeigt, wie diese Enzymreaktion mit Hilfe der BioChem-Bibliothek in OMEdit umgesetzt werden kann.

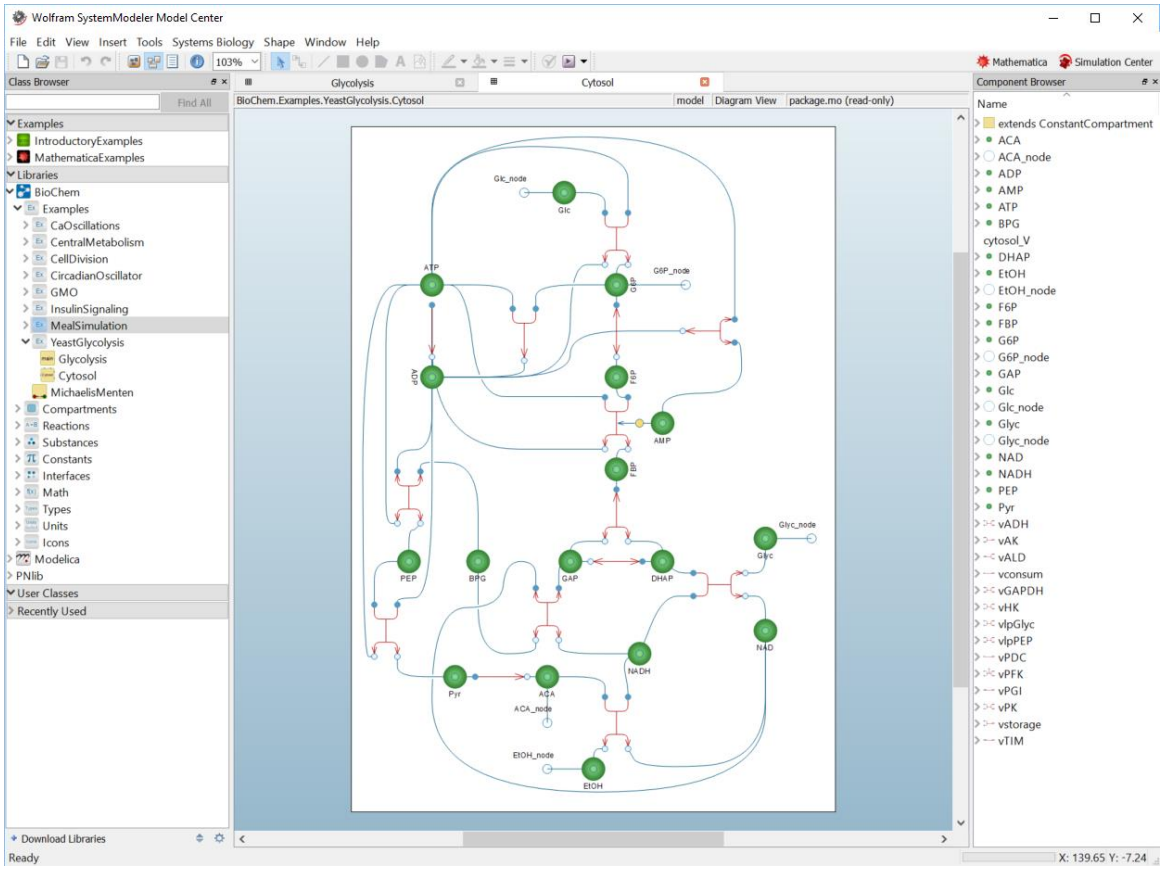

Abbildung 3.2: Das Beispiel BioChem.Examples.YeastGlycolysis.Cytosol der Bibliothek BioChem 1.2 im Wolfram SystemModeler.

Hierfür werden lediglich drei Komponenten benötigt: die zwei Substanzen G6P und F6P, sowie die Reaktion selbst. In den Komponenten für die Substanzen werden die Startkonzentrationen gesetzt und in der Komponente für die Reaktion werden die vier charakteristischen Michaelis-Menten-Parameter (siehe Abschnitt 2.3.3) definiert.

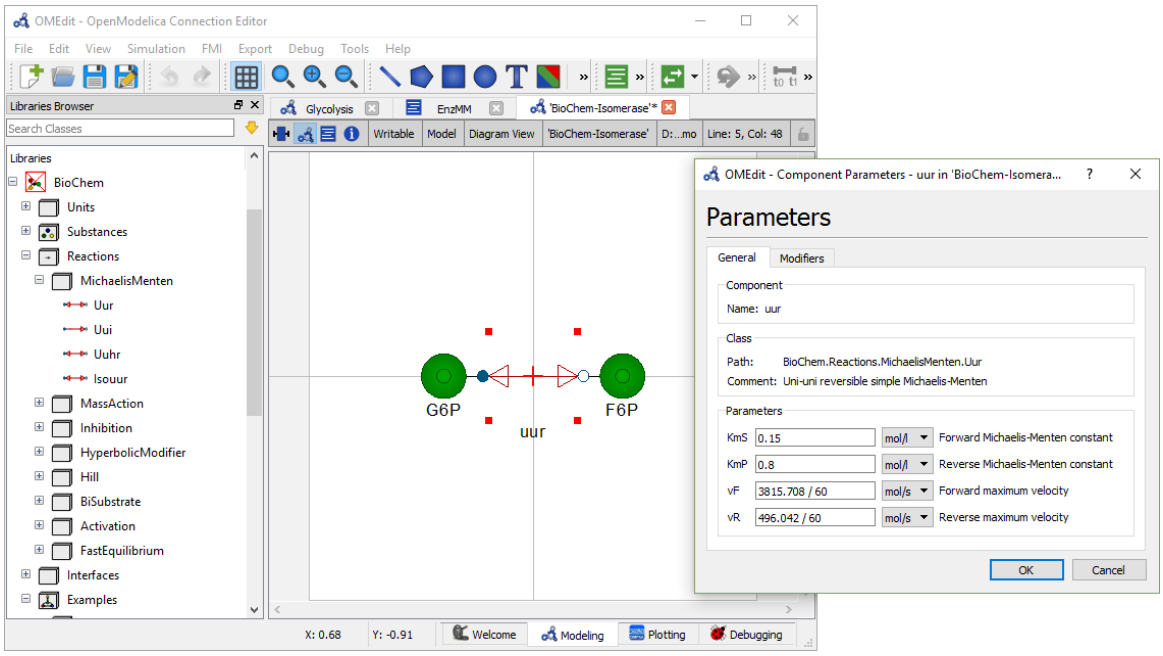

Abbildung 3.3: Das Beispiel der Glucose-6-phosphat-Isomerase, umgesetzt mit der Bibliothek BioChem, in OMEdit.

Die entsprechende textuelle Modellbeschreibung, ohne grafischen Annotationen, ist im nachstehenden Listing 3.1 dargestellt.

```
1  model 'BioChem-Isomerase'
2    "Enzym-Reaktion mit Michaelis-Menten Kinetik"
3    extends BioChem.Compartments.Compartment;
4    BioChem.Substances.Substance F6P(c.start=0.5)
5      "Fructose-6-phosphate";
6    BioChem.Substances.Substance G6P(c.start=4.1)
7      "Glucose-6-phosphate";
8    BioChem.Reactions.MichaelisMenten.Uur uur(
9      vF=3815.708/60, vR=496.042/60, KmS=0.15, KmP=0.8);
10 equation
11   connect(G6P.n1, uur.s1);
12   connect(uur.p1, F6P.n1);
13 end 'BioChem-Isomerase';
```

Listing 3.1: Textuelle Modellrepräsentation der Glucose-6-phosphat-Isomerase auf Basis der BioChem-Bibliothek.

Der BioChem-Bibliothek liegt kein Petri-Netz-Formalismus zugrunde, sondern die biologischen Netzwerke werden direkt in differential-algebraische Gleichungssysteme überführt. Hierfür werden Konnektoren definiert, die aus einer Potentialvariable für die Stoffkonzentration (siehe Zeile 19 und 31 in Listing 3.2) und einer Flussvariablen für den Massenstrom (siehe Zeile 20 und 32 in Listing 3.2) bestehen. Die Modelle für eine Reaktion beschreiben die algebraischen Beziehungen zur Berechnung des Massenstroms (siehe Zeile 84 in Listing 3.2). Die Modelle für die Substanzen enthalten die Differentialgleichung, die die Änderung der Stoffkonzentration (siehe Zeile 71 in Listing 3.2) über die Flussvariable des entsprechenden Konnektors beschreibt.

```
1  package BioChem.Units "Units used in BioChem"
2    type ReactionRate = Real(quantity="Reaction rate", unit="mol/s");
3    type Concentration = Real(quantity="Concentration", unit="mol/l", min=0);
4    type Volume = Modelica.SIunits.Conversions.NonSIunits.Volume_litre;
5    type MolarFlowRate = Real(quantity="Molar flow rate", unit="mol/s");
6  end Units;
7
8  partial model BioChem.Interfaces.Reactions.Basics.Reaction
9    "Basics for a reaction edge"
10   BioChem.Units.ReactionRate rr "Rate of the reaction";
11 end Reaction;
12
13 model BioChem.Interfaces.Reactions.Basics.OneSubstrateReversible
14   BioChem.Interfaces.Nodes.SubstrateConnector s1;
15 end OneSubstrateReversible;
16
17 connector BioChem.Interfaces.Nodes.SubstrateConnector
18   "Connector between substances and reactions (substrate side of reaction)"
19   BioChem.Units.Concentration c;
20   flow BioChem.Units.MolarFlowRate r;
21   input BioChem.Units.Volume V;
22 end SubstrateConnector;
23
24 partial model BioChem.Interfaces.Reactions.Basics.OneProduct
```

```
25    "SubstanceConnector for one product"
26    BioChem.Interfaces.Nodes.ProductConnector p1;
27  end OneProduct;
28
29  connector BioChem.Interfaces.Nodes.ProductConnector
30    "Connector between substances and reactions (product side of reaction)"
31    BioChem.Units.Concentration c;
32    flow BioChem.Units.MolarFlowRate r;
33    input BioChem.Units.Volume V;
34  end ProductConnector;
35
36  partial model BioChem.Interfaces.Reactions.Uur
37    "Uni-Uni reversible reaction"
38    extends BioChem.Interfaces.Reactions.Basics.Reaction;
39    extends BioChem.Interfaces.Reactions.Basics.OneSubstrateReversible;
40    extends BioChem.Interfaces.Reactions.Basics.OneProduct;
41    BioChem.Units.StoichiometricCoefficient nS1=1
42      "Stoichiometric coefficient for the substrate";
43    BioChem.Units.StoichiometricCoefficient nP1=1
44      "Stoichiometric coefficient for the product";
45  equation
46    s1.r = nS1*rr;
47    p1.r = -nP1*rr;
48  end Uur;
49
50  partial model BioChem.Interfaces.Substances.Substance
51    "Basics for a substance"
52    BioChem.Units.Concentration c(stateSelect=StateSelect.prefer)
53      "Current concentration of substance (mM)";
54    BioChem.Units.MolarFlowRate rNet
55      "Net flow rate of substance into the node";
56    BioChem.Units.AmountOfSubstance n(stateSelect=StateSelect.prefer)
57      "Number of moles of substance in pool (mol)";
58    BioChem.Interfaces.Nodes.SubstanceConnector n1;
59  protected
60    outer BioChem.Units.Volume V "Compartment volume";
61  equation
62    rNet = n1.r;
63    c = n1.c;
64    V = n1.V;
65    c = n/V;
66  end Substance;
67
68  model BioChem.Substances.Substance "Substance with variable concentration"
69    extends BioChem.Interfaces.Substances.Substance;
70  equation
71    der(n) = rNet;
72  end Substance;
73
74  model BioChem.Reactions.MichaelisMenten.Uur
75    "Uni-uni reversible simple Michaelis-Menten"
76    extends BioChem.Interfaces.Reactions.Uur;
77    parameter BioChem.Units.Concentration KmS=1
78      "Forward Michaelis-Menten constant";
79    parameter BioChem.Units.Concentration KmP=1
80      "Reverse Michaelis-Menten constant";
```

```
81    parameter BioChem.Units.ReactionRate vF=1 "Forward maximum velocity";
82    parameter BioChem.Units.ReactionRate vR=1 "Reverse maximum velocity";
83  equation
84    rr = (vF*s1.c/KmS - vR*p1.c/KmP)/(1 + s1.c/KmS + p1.c/KmP);
85  end Uur;
```

Listing 3.2: Definition einer Michaelis-Menten Gleichung in der Bibliothek BioChem mit allen Abhängigkeiten.

Mit diesem Ansatz können beispielsweise, abhängig von der Modellierung, Stoffkonzentrationen negativ werden oder unidirektionale Reaktionen rückwärts ablaufen. Zudem unterstützt dieser Ansatz nur rein kontinuierliche Netzwerke.

Derzeit ist die BioChem-Bibliothek in der Version 1.0.1 zur freien Nutzung unter der *Modelica License 2* verfügbar. Das Unternehmen Wolfram bietet zusätzlich die weiterentwickelte Version 1.2 unter der *Mozilla Public License* auf seiner Webseite[1] an.

3.2 PNlib 1.0

Die Modelica Bibliothek PNlib [26] ermöglicht die Modellierung von Petri-Netzen auf Basis des xHPN-Formalismus. Dieser Formalismus verbindet kontinuierliche, diskrete, stochastische, funktionale und kapazitive Petri-Netze. Zusätzlich sind noch spezielle Kanten in Form der Hemmkante, Testkante und Lesekante verfügbar. Die Modellierung erfolgt grafisch durch das Verbinden der entsprechenden Komponenten, die in Abbildung 3.4 dargestellt sind.

Die Bibliothek wurde auf der 9. Internationalen Modelica-Konferenz 2012 in München erstmals vorgestellt und gewann dort den 1. Preis des *Library Award* für die beste freie Modelica-Bibliothek.

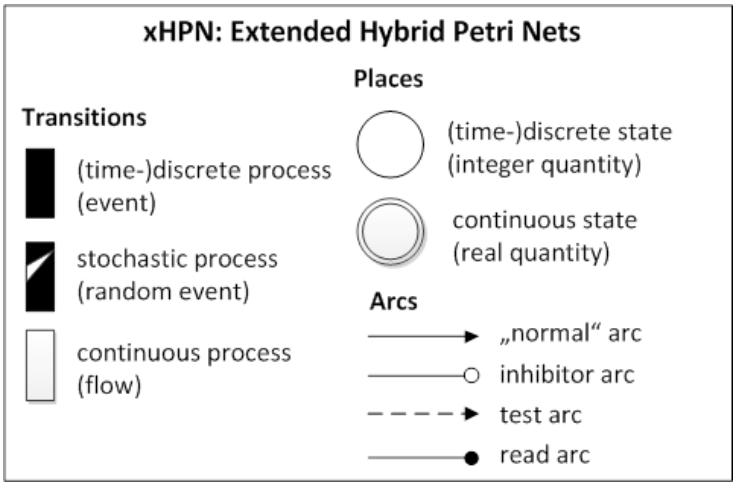

Abbildung 3.4: Elemente des xHPN-Formalismus, die von der Bibliothek PNlib bereitgestellt werden.

In dem Kapitel „Grundlagen“ wurde bereits gezeigt, wie sich Enzymreaktionen und konkret die Glucose-6-phosphat-Isomerase mit Petri-Netzen umsetzen lässt. Dies lässt sich direkt auf die grafische Modellierung mit der PNlib übertragen, wie in Abbildung 3.5 zu sehen ist.

[1] https://www.wolfram.com/system-modeler/libraries/biochem/

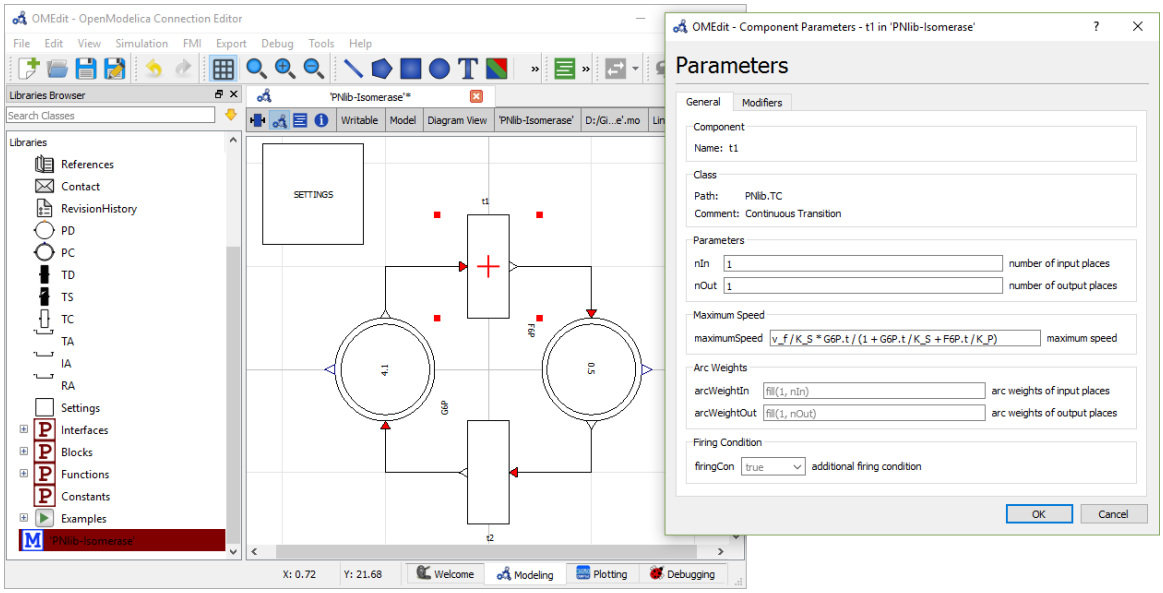

Abbildung 3.5: Das Beispiel der Glucose-6-phosphat-Isomerase, umgesetzt mit der Bibliothek PNlib, in OMEdit.

Die textuelle Darstellung des obigen Beispiels ist in Listing 3.3 dargestellt. Ähnlich zu dem BioChem-Beispiel des vorherigen Abschnitts werden die Substanzen mit ihren Startkonzentrationen definiert. Bei der Reaktion gibt es einen wesentlichen Unterschied: Transitionen haben immer einen definierten Vor- und Nachbereich und können daher nur unidirektional feuern, wohingegen in dem BioChem-Beispiel eine reversible Reaktion verwendet wurde. Diese muss in dem Petri-Netz durch zwei Transitionen modelliert werden, die getrennt voneinander die Hin- und Rückreaktionen abbilden.

```
1  model 'PNlib-Isomerase'
2    PNlib.PC G6P(nIn=1, nOut=1, startMarks=4.1);
3    PNlib.PC F6P(nIn=1, nOut=1, startMarks=0.5);
4    PNlib.TC t1(maximumSpeed = v_f / K_S * G6P.t /
5      (1 + G6P.t / K_S + F6P.t / K_P), nIn = 1, nOut = 1);
6    PNlib.TC t2(maximumSpeed = v_r / K_P * F6P.t /
7      (1 + G6P.t / K_S + F6P.t / K_P), nIn = 1, nOut = 1);
8    inner PNlib.Settings settings;
9
10   parameter Real v_f(final unit="mM/s") = 3815.708/60;
11   parameter Real v_r(final unit="mM/s") = 496.042/60;
12   parameter Real K_S(final unit="mM") = 0.15;
13   parameter Real K_P(final unit="mM") = 0.8;
14 equation
15   connect(G6P.outTransition[1], t1.inPlaces[1]);
16   connect(t2.outPlaces[1], G6P.inTransition[1]);
17   connect(t1.outPlaces[1], F6P.inTransition[1]);
18   connect(F6P.outTransition[1], t2.inPlaces[1]);
19 end 'PNlib-Isomerase';
```

Listing 3.3: Textuelle Modellrepräsentation der Glucose-6-phosphat-Isomerase auf Basis der Bibliothek PNlib.

Die Modellierung von Petri-Netzen mit Modelica hat einen gravierenden Nachteil. Kantengewichte, die ein zentraler Bestandteil von Petri-Netzen sind, sind in Modelica nicht

vorgesehen. Daher wurde bei dem Design der PNlib entschieden, die Kantengewichte als Arrays in den Transitionen zu speichern. Dies bedeutet allerdings, dass herkömmliche grafische Modelica-Editoren, wie z.B. OMEdit, diese Informationen nicht auf den Kanten anzeigen. Als zusätzlicher Nachteil gilt, dass beim Erstellen und Editieren der Netze, die Indizes der Kanten manuell vom Modellierer auf die Arrays der Kantengewichte bezogen werden müssen. Dies ist umständlich und fehleranfällig. Daher ist es ratsam, eine spezielle grafische Oberfläche für die Arbeit mit der PNlib, wie z.B. VANESA, zu verwenden.

3.3 VANESA

VANESA [27] ist ein grafischer Editor zum Erstellen von biologischen Netzwerken, der seit 2009 an der Universität Bielefeld entwickelt[1] wird. Der Name ist ein Akronym und leitet sich aus dem englischen „**v**isualization and **a**nalysis of **ne**tworks in **s**ystems biology **a**pplications“ ab. Das Werkzeug unterstützt den Anwender beim Untersuchen von systembiologischen Fragestellungen und Hypothesen. Die Modellierung der Netzwerke kann durch manuelle Eingabe von experimentell erhobenen Daten oder Literaturwerten geschehen. Eine weitere Option der Modellierung bietet das angebundene Data Warehouse DAWIS-M.D. [28]. Das Data Warehouse beinhaltet die Integration mehrerer molekularbiologischer Datenbanken, wie z.B. KEGG, BRENDA, Mint und IntAct. Durch den implementierten Webservice lassen sich KEGG Pathways abrufen und Tiefensuchen auf Proteinen und Metaboliten durchführen.

Wie bereits in den Grundlagen eingeführt, stellen Petri-Netze eine weitere Möglichkeit der Modellierung biologischer Netze dar. VANESA erlaubt die Modellierung auf Basis des xHPN-Formalismus. Zur Simulation wird das Petri-Netz als Modelica-Modell exportiert und mit Hilfe der PNlib und dem OpenModelica-Compiler simuliert. Die generierte Simulation verbindet sich über eine TCP/IP-Verbindung mit VANESA und übermittelt so die Simulationsergebnisse on-the-fly.

In Abbildung 3.6 ist das Modell der Glucose-6-phosphat-Isomerase als Petri-Netz in VANESA 2.0 dargestellt. In älteren Versionen gab es einige Einschränkungen bezüglich des xHPN-Formalismus. Diese bestanden unter anderem darin, dass kontinuierliche Transitionen immer die maximale Geschwindigkeit von 1 zugewiesen bekamen. Daher mussten individuelle Reaktionsgeschwindigkeiten über funktionale Kanten realisiert werden. Zudem konnten noch keine Parameter definiert werden, weshalb diese in die Kantengewichte direkt eingesetzt werden mussten. In den Kantengewichten konnte lediglich durch die ID (z.B. P1001) auf Plätze zugegriffen werden, und nicht über den Namen (z.B. G6P), wodurch die Lesbarkeit und Arbeit erschwert wurde. Im Rahmen des MoRitS[2]-Projektes wurde intensiv an

[1] VANESA ist auf SourceForge gehostet:
⇒ https://sourceforge.net/projects/vanesa/

[2] Die Forschungskooperation „Modellbasierte Realisierung intelligenter Systeme in der Nano- und Biotechnologie“ (Förderungsnummer: 321 - 8.03.04.03 - 2012/02) wurde vom Ministerium für Innovation, Wissenschaft und Forschung des Landes Nordrhein-Westfalen (MIWF NRW) gefördert.

VANESA gearbeitet, sodass in der aktuellen Version 2.0 der xHPN-Formalismus vollständig unterstützt wird und die angesprochenen Unzulänglichkeiten beseitigt wurden.

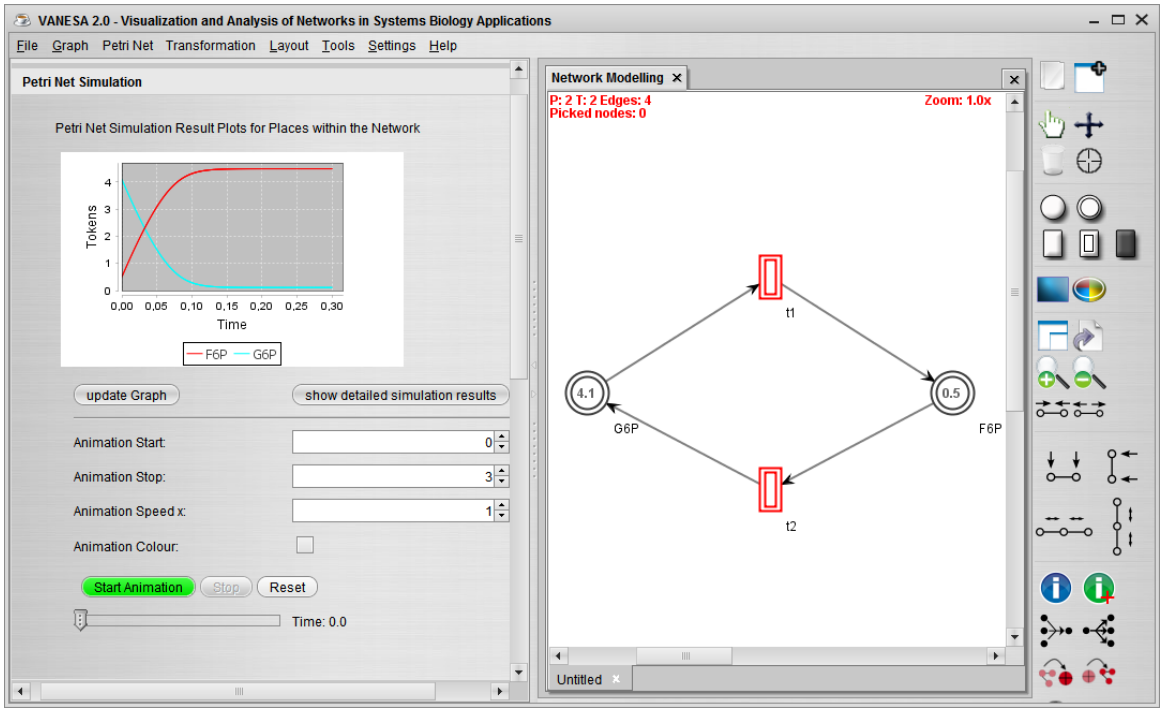

Abbildung 3.6: Das Beispiel der Glucose-6-phosphat-Isomerase mit der aktuellen Version 2.0 von VANESA.

Da sowohl biologische Netze als auch Petri-Netze Graphen sind, lassen sich beide Netzklassen mit bekannten graphtheoretischen Algorithmen untersuchen. Dazu bietet VANESA verschiedene Methoden [27]. Darüber hinaus sind spezifische Analysemethoden für diskrete Petri-Netze vorhanden, wie das Berechnen des Überdeckungsgraphen und des Erreichbarkeitsgraphen.

Liegen Experimentdaten in Form von Microarraydaten zu einem modellierten Netzwerk vor, können diese auf die Knoten gemappt werden. Die Farbe des Knotens repräsentiert dann die gemessene Expression.

3.4 Snoopy

In [29] wird Snoopy als Petri-Netz Programm beschrieben, das auf viele unterschiedliche Anwendungen in der synthetischen Biologie und Systembiologie ausgerichtet ist. Es werden Petri-Netze der beiden Oberklassen ungefärbte Petri-Netze und gefärbte Petri-Netze unterstützt. Zu beiden Konzepten sind eine Reihe spezialisierter Petri-Netz-Klassen verfügbar. Modelle können innerhalb ihrer Oberklasse zwischen den spezialisierten Petri-Netz-Klassen umgewandelt werden. Dies erfolgt teilweise automatisch und teilweise halb-manuell. Modelle können hierarchisch strukturiert werden, um das Arbeiten mit großen Netzen zu vereinfachen. Unterschiedliche Petri-Netz-Klassen können gleichzeitig verwendet werden, wobei das Programm die grafische Oberfläche automatisch an die aktive Klasse anpasst. Diese Netze können simuliert und für unterschiedliche Analysewerkzeuge exportiert werden. Durch den generischen Aufbau kann Snoopy um weitere Petri-Netz-Klassen erweitert werden.

Snoopy ermöglicht die Modellierung kontinuierlicher Petri-Netze. Diese können Plätze, Transitionen, Kanten, Testkanten, usw. enthalten. Plätzen dürfen ausschließlich nicht-negative Markierungen zugewiesen werden. Den Geschwindigkeitsfunktionen der kontinuierlichen Transitionen können von den Plätzen im Vorbereich abhängen. Zudem können Konstanten definiert werden, auf die alle Elemente des Petri-Netzes Zugriff haben (siehe Abbildung 3.7, rechts). Dies erleichtert die Modellierung der Geschwindigkeits-funktionen deutlich und erhöht die Übersichtlichkeit.

Snoopy bietet zudem die Möglichkeit, Simulationsergebnisse direkt anzuzeigen und diese zu exportieren.

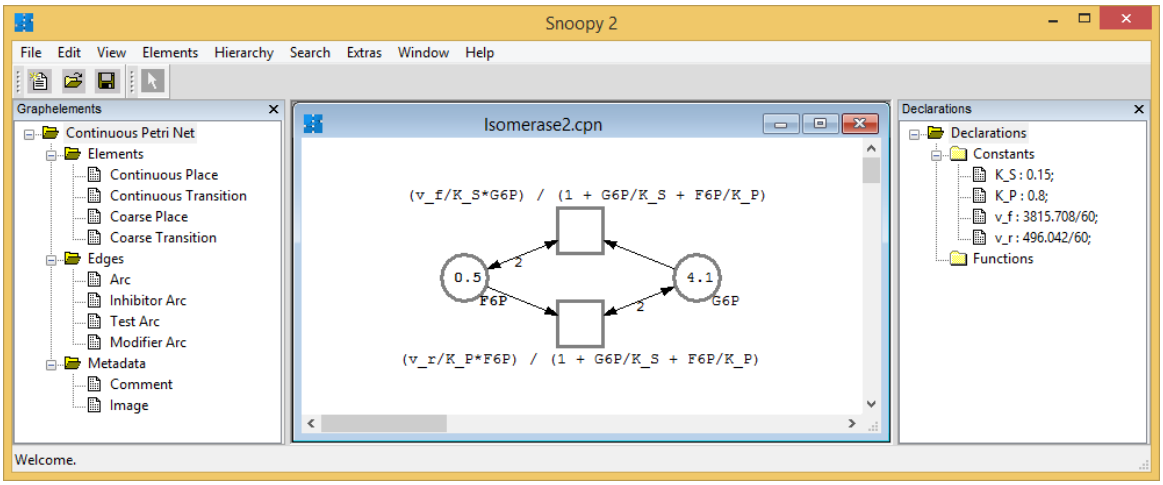

Abbildung 3.7: Das Beispiel Glucose-6-phosphat-Isomerase in Snoopy.

Wie bei den zuvor beschriebenen Modellierungswerkzeugen wird auch hier exemplarisch die Glucose-6-phosphat-Isomerase umgesetzt. Das entsprechende reversible Modell kann als Petri-Netz, wie z.B. in Abbildung 2.13 gezeigt, modelliert werden. Die Umsetzung mit Snoopy ist in Abbildung 3.7 zu sehen. Das dort gezeigte Petri-Netz unterscheidet sich durch die Kanten von den zuvor gezeigten Umsetzungen. Dies liegt daran, dass Snoopy ohne weiteres nur die Plätze des Vorbereichs als Größe in die Geschwindigkeitsgleichung der Transitionen zulässt. Die Kinetik der Isomerase hängt in diesem Beispiel allerdings sowohl von $\mathrm{G6P}$ wie auch $\mathrm{F6P}$ ab, die sich jeweils in Vor- und Nachbereich aufteilen. Die Einschränkung von Snoopy kann umgangen werden, indem die Plätze der Nachbereiche der Transitionen mit dieser jeweils durch zwei Kanten verbunden werden: Die erste Kante geht zur Transition mit Kantengewicht 1. Die zweite Kante geht zum Platz mit Kantengewicht 2. So ergeben beide Kanten zusammengenommen eine Kante, von Transition zum Platz, mit einem effektiven Kantengewicht von 1.

Für die Simulation wird das kontinuierliche Petri-Netz von Snoopy in ein System gewöhnlicher Differentialgleichungen überführt. Hierfür wird für jeden Platz ein Zustand angelegt, dessen Ableitung die Summe der Geschwindigkeitsfunktionen aller verbundenen Transitionen zugewiesen wird. Die Geschwindigkeitsfunktionen werden zuvor mit dem entsprechenden Kantengewicht multipliziert und im Falle von ausgehenden Kanten wird die

Geschwindigkeitsfunktion negiert. Das so konstruierte Gleichungssystem kann von Snoopy in Textform exportiert werden, wie in Listing 3.4 gezeigt. Das Petri-Netz kann also als grafische Veranschaulichung des zugehörigen DGL-Systems interpretiert werden.

```
1 dF6P/dt =   (v_f/K_S*G6P)/(1+G6P/K_S+F6P/K_P)
2           -(v_r/K_P*F6P)/(1+G6P/K_S+F6P/K_P)
3 dG6P/dt =   (v_r/K_P*F6P)/(1+G6P/K_S+F6P/K_P)
4           -(v_f/K_S*G6P)/(1+G6P/K_S+F6P/K_P)
```

Listing 3.4: Das zum Beispiel der Glucose-6-phosphat-Isomerase aus Abbildung 3.7 zugehörige DGL-System.

Dies bedeutet, dass die Richtung der Kanten während der Simulation keine Bedeutung haben und Tokens in beliebiger Richtung über die Kanten „laufen“ können. Somit ist es auch möglich, dass Markierungen während der Simulation negativ werden. All dies zeigt, dass Snoopy im Grunde genommen nicht als Simulator für kontinuierliche Petri-Netze (vgl. [1], [30], Kapitel „Grundlagen“) angesehen werden kann.

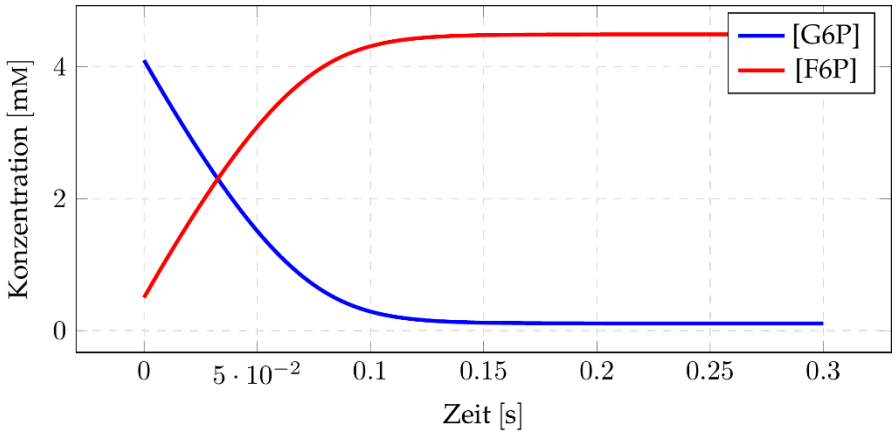

Abbildung 3.8: Konzentrationsverläufe zum Beispiel Glucose-6-phosphat-Isomerase.

Das Beispiel aus Abbildung 3.7 kann mit Snoopy noch weiter ad absurdem geführt werden und, wie in Abbildung 3.9 zu sehen, mit lediglich einer Transition umgesetzt werden. In diesem Fall ist der Platz F6P lediglich im Vorbereich der Transition und dennoch steigt die Markierung dieses Platzes während der Simulation, wie die entsprechenden Simulationsergebnisse in Abbildung 3.8 zeigen.

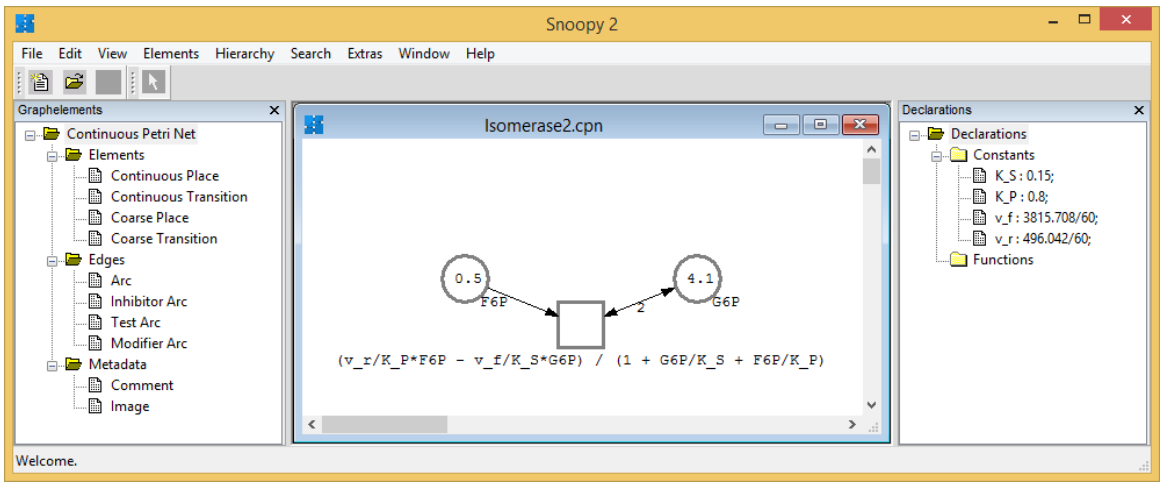

Abbildung 3.9: Beispiel der reversiblen Glucose-6-phosphat-Isomerase mit lediglich einer Transition in Snoopy.

Das entsprechende DGL-System ist in Listing 3.5 dargestellt und unterscheidet sich zu dem System aus Listing 3.4 nur durch die Klammerung der Ausdrücke; mathematisch sind beide Systeme identisch.

```
1 dF6P/dt = -(v_r/K_P*F6P-v_f/K_S*G6P)/(1+G6P/K_S+F6P/K_P)
2 dG6P/dt =  (v_r/K_P*F6P-v_f/K_S*G6P)/(1+G6P/K_S+F6P/K_P)
```

Listing 3.5: DGL-System zum Beispiel Glucose-6-phosphat-Isomerase aus Abbildung 3.9.

3.5 Cell Illustrator

Cell Illustrator [31] ist ein Programm, das sich speziell an Systembiologen richtet. Es erlaubt, komplexe biologische Prozesse abzubilden und zu simulieren. Es basiert auf dem Programm Genomic Object Net (kurz GON) [32], das zu Forschungszwecken an der Universität Tokio entwickelt wurde.

Der Cell Illustrator basiert auf einem Formalismus für hybride funktionale Petri-Netze (kurz HFPN) [33], [34] und erlaubt damit unter anderem das Modellieren von metabolischen Pathways, Signaltransduktionskaskaden und einer Vielzahl von dynamischen Interaktionen biologischer Objekte, wie z.B. genomic DNA, mRNA und Proteinen. Es wird dazu eingesetzt, biologische Pathways darzustellen, Laborversuche zu analysieren und Hypothesen zu testen. Zusätzlich unterstützt es die Arbeit durch Modelldiagramme und graphisch aufbereitete Simulationsergebnisse.

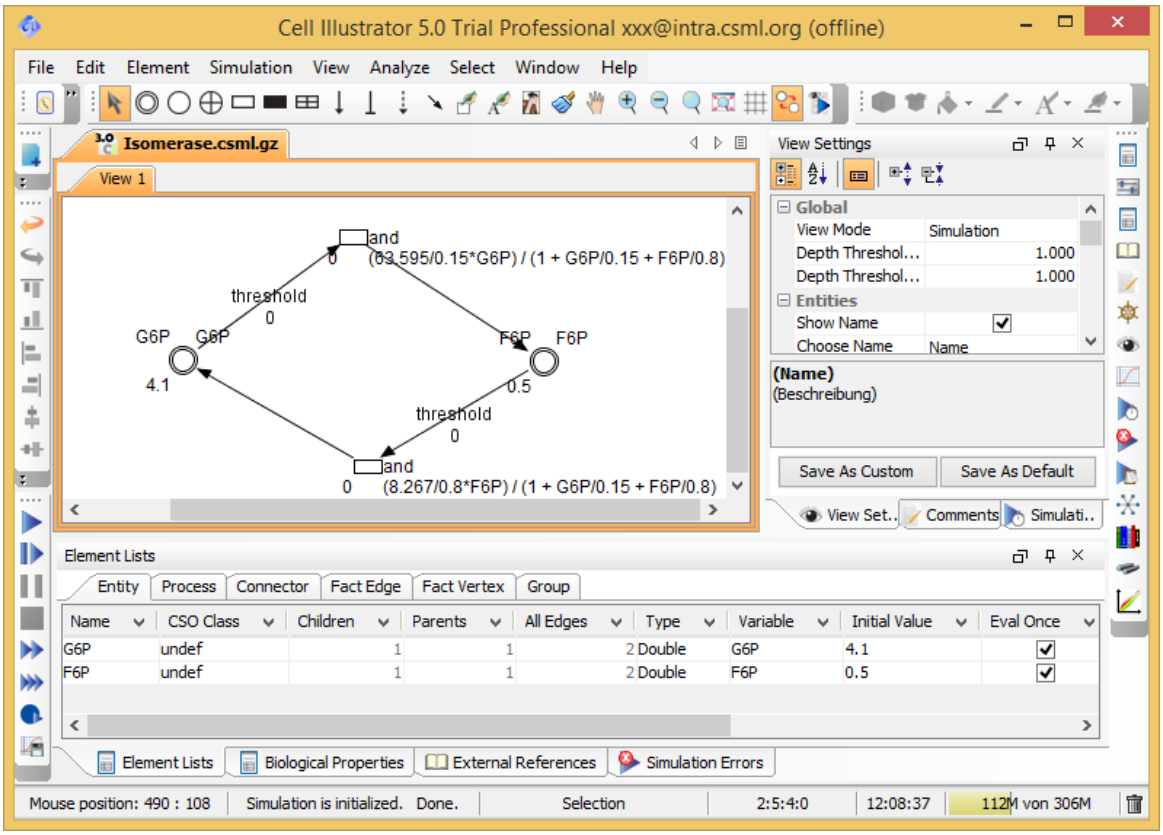

Abbildung 3.10: Umsetzung des Beispiels der Glucose-6-phosphat-Isomerase mit dem Cell Illustrator 5.

Das Softwarepaket ist sehr mächtig. Es verhält sich allerdings im Wesentlichen wie eine Blackbox. Daher ist es nicht möglich, in den Simulationsprozess hineinzuschauen, um z.B. die Logik, mit der Konflikte aufgelöst werden, zu verstehen. Entsprechend kann der Formalismus auch nicht erweitert werden.

3.6 CPN Tools

CPN Tools [18] ist ein umfassendes Programm zum Modellieren, Simulieren und Analysieren von gefärbten Petri-Netzen.

Ursprünglich wurde das Programm von der Arbeitsgruppe Coloured Petri Nets[1] (kurz CPN) an der Aarhus Universität zwischen 2000 und 2010 entwickelt. Hierbei waren primär Kurt Jensen, Søren Christensen, Lars M. Kristensen und Michael Westergaard involviert. Seit Herbst 2010 wird das Programm in der Arbeitsgruppe AIS (Architecture of Information Systems) an der Eindhoven University of Technology in den Niederlanden betreut.

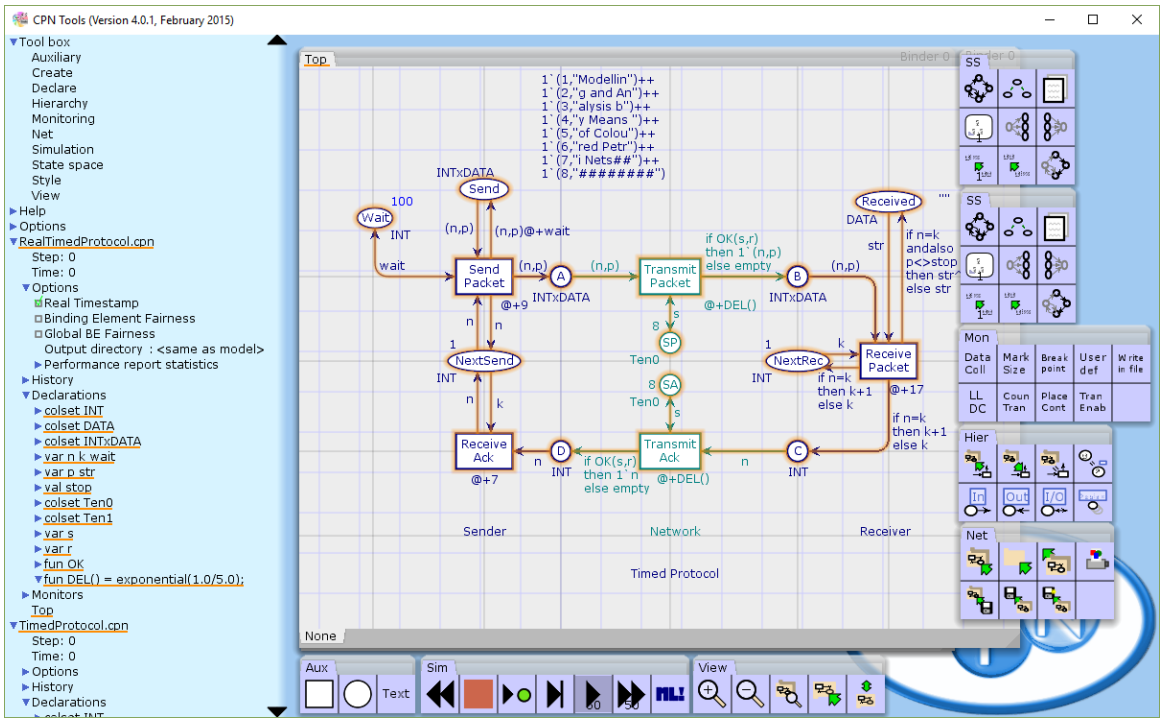

Abbildung 3.11: CPN Tools mit dem Standardbeispiel "TimedProtocol".

Der Petri-Netz-Formalismus hinter CPN Tools basiert auf gefärbten Petri-Netzen [35] und einem diskreten Zeitkonzept [36], welches den Tokens Zeitpunkte zuweist. Diese können durch das Feuern von Transitionen verändert werden. Dies wird mit dem Beispiel aus Abbildung 3.12 verdeutlicht.

[1] Die Homepage der Arbeitsgruppe CPN: http://cs.au.dk/CPnets/

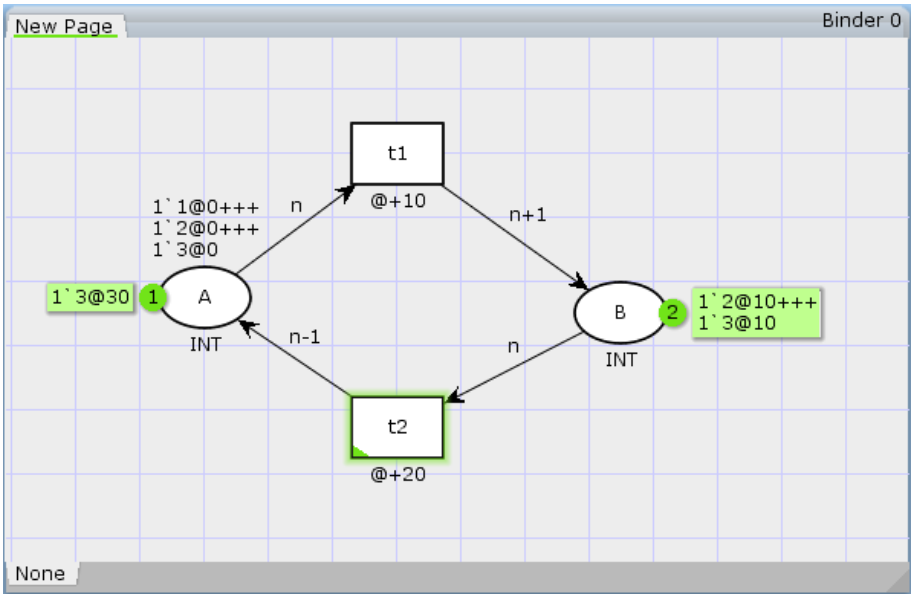

Abbildung 3.12: Minimalbeispiel eines gefärbten Petri-Netzes mit CPN Tools.

Das Modell aus Abbildung 3.12 besteht aus den beiden Plätzen A und B, die beide der Klasse `INT` angehören. Die Klasse `INT` umfasst die ganzen Zahlen, sodass jedem Token ein ganzzahliger Wert zugeordnet ist. Zudem besteht das Modell noch aus den beiden Transitionen t1 und t2. Im Ausgangszustand, der oberhalb der Plätze dargestellt ist, enthält Platz A drei Tokens, denen die Werte 1, 2 und 3 zugeordnet sind und Platz B keine Tokens. Die Tokens werden als Menge und die Menge der Tokens eines Platzes als Multimenge dargestellt. Der `+++`–Operator ist als Multimengenvereinigung zu verstehen. Zudem besitzen hier im Ausgangszustand alle Tokens den Zeitwert 0. Die Zeitwerte werden durch das `@`–Zeichen gekennzeichnet. Der aktuelle Zustand wird in CPN Tools in grünen Kästchen neben den Plätzen angezeigt. Platz A enthält hier ein Token mit Wert 3 und Zeitwert 30. Platz B enthält zwei Tokens mit den Werten 2 und 3 und jeweils dem Zeitwert 10.

Der Übergang zu dem Zustand aus Abbildung 3.13 erfolgt durch das Feuern von Transition t2. Hierbei wird zunächst der Modus bestimmt. Dieser kann explizit oder Zufall-basiert ausgewählt werden. Hier wurde der Modus $n = 3$ gewählt. Dadurch wird das Token mit dem Wert 3 aus dem Platz B entfernt und ein neues Token mit dem Wert $n - 1 = 2$ zu Platz A hinzugefügt. Der Zeitstempel des entfernten Tokens wird hierbei ausgelesen, und um den Wert 20 erhöht. Dies zeigt, dass sowohl die Tokens als auch das Zeitkonzept diskret sind und sich in einem Zustand des Petri-Netzes Tokens mit unterschiedlichen Zeitstempeln befinden können.

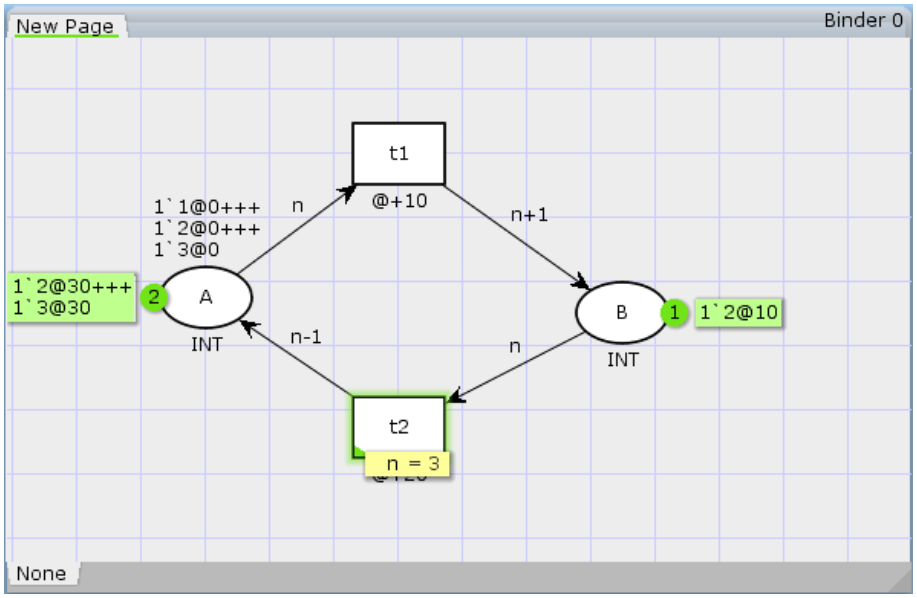

Abbildung 3.13: Der neue Zustand des Petri-Netzes aus Abbildung 3.12, nachdem t2 im Modus n=3 gefeuert hat.

Der Formalismus und das Programm CPN Tools eignen sich z.B. für die Simulation von Datenprotokollen (siehe Abbildung 3.11). Das verwendete Zeitkonzept ist grundverschieden zu den gezeiteten Petri-Netzen, wie sie zuvor in den Grundlagen eingeführt wurden. Somit können kontinuierliche Prozesses, wie Enzymreaktionen, mit CPN Tools nicht sinnvoll umgesetzt werden.

3.7 Zusammenfassung

In diesem Kapitel wurde eine Reihe von Programmen und Bibliotheken vorgestellt, die entweder für die Arbeit mit Petri-Netzen oder aber direkt für die Systembiologie entwickelt wurden. Sofern möglich, wurde mit jedem Werkzeug das Beispiel der Glucose-6-phosphat-Isomerase umgesetzt, um die Unterschiede und Besonderheiten der einzelnen Ansätze herauszustellen. Die Tabelle 3.1 fasst die wesentlichen Unterschiede aller hier vorgestellten Programme und Bibliotheken zusammen.

Tabelle 3.1: Vergleich der im Kapitel „Verwandte Arbeiten“ vorgestellten Werkzeuge zur Simulation biologischer Netzwerke.

Name	BioChem	PNlib	VANESA	Snoopy	Cell Illustrator	CPN Tools
Typ	Bibliothek	Bibliothek	Editor	Simulator	Simulator	Simulator
Transparente Implementierung	Ja	Ja	Ja	Nein	Nein	Nein
Physikalische Einheiten	Ja	Ja	Ja	Nein	Nein	Nein
Toolunabhängige Simulation	Ja	Ja	Ja	Nein	Nein	Nein
Formalismus	ODE/DAE	Petri-Netz (xHPN)	Petri-Netz (xHPN)	Petri-Netz (diverse)	Petri-Netz (HFPN)	Petri-Netz (Timed CPN)
Lizenz	Frei	Frei	Frei	Frei	Kommerziell	Frei

Die Bibliothek **BioChem** ist dafür entwickelt worden, biologische Prozesse anschaulich mit einem System von gewöhnlichen Differentialgleichungen abzubilden. Da die Bibliothek in Modelica geschrieben ist, ist ihre Umsetzung transparent und somit nachvollziehbar. Zudem kann leicht in die Umsetzung eingegriffen und diese nach Belieben angepasst werden. Da die Bibliothek in der Modellierungssprache Modelica geschrieben ist, werden physikalische Einheiten für alle Modellgrößen unterstützt. Das Erstellen und Simulieren von Modellen ist an kein spezielles Programm gebunden und kann im Grunde mit jeder Modelica-Umgebung erfolgen.

Die Modelica-Bibliothek **PNlib** ist zum Erstellen von zeitbasierten Petri-Netzen (xHPN) entwickelt worden. In Verbindung mit einem Modelica-Compiler, können die Modelle auch simuliert werden. Ein großer Vorteil ist, dass die Simulation nicht an eine spezielle Simulationsumgebung gebunden ist, und sowohl mit kommerziellen als auch quelloffenen und freiverfügbaren Umgebungen erfolgen kann. Durch die transparente Umsetzung mittels Modelica-Quellcode lässt sich der umgesetzte Formalismus leicht nachvollziehen und individuell anpassen und ergänzen. Genau wie bei der BioChem-Bibliothek, lassen sich physikalische Einheiten für alle Modellgrößen definieren, wodurch eine symbolische Fehlerprüfung auf Basis dieser zusätzlichen Informationen durchgeführt werden kann.

Das Werkzeug **VANESA** ist ein mächtiges Werkzeug für die Arbeit mir biologischen Netzen und Petri-Netzen. Es unterstützt den Anwender beim Erstellen, Kombinieren, Analysieren und Simulieren solcher Netzwerke. Die Netze lassen sich mittels standardisierter Formate mit anderen Programmen austauschen. Durch die Integration von OpenModelica in Verbindung mit der PNlib ist auch eine transparente Simulation der Petri-Netze möglich.

Das Petri-Netz-Programm **Snoopy** unterstützt eine ganze Reihe unterschiedlicher Petri-Netz-Klassen. Zeitbasierte kontinuierliche Petri-Netze, wie sie essentiell für die Umsetzung biologischer Prozesse sind, werden jedoch nur ungenügend unterstützt.

Der **Cell Illustrator** ist ein kommerzieller Simulator für unterschiedliche biologische Prozesse. Es basiert auf einem Petri-Netz-Formalismus und kann auch direkt für die Modellierung und Simulation von Petri-Netzen eingesetzt werden. Die Simulation kann jedoch nicht im Detail nachvollzogen werden, und es bleibt unklar, wie diese tatsächlich abläuft. Zudem lässt sich weder der Formalismus noch die Ausführung anpassen.

Das Programm **CPN Tools** ist zum Erstellen und Simulieren zeitbasiertet gefärbter Petri-Netze konzipiert. Allerdings sind sowohl die Tokens, als auch das Zeitkonzept diskret, weshalb sie sich nicht für die Darstellung kontinuierlicher biologischer Prozesse eignen. Das Konzept der Färbung hingegen, ist auch für den biologischen Einsatz interessant. In Anlehnung hieran wird im Weiteren das Konzept gefärbter Plätze mit dem xHPN-Formalismus umgesetzt, um den Informationsgehalt und die Aussagekraft der Modelle zu erhöhen.

4 Neuentwurf und Implementierung des OpenModelica-Backend

Die Aufgabe des OpenModelica-Backend ist es, das flache Modell durch mathematische Transformationen für die Codegenerierung und somit für die Berechnung des Systemverhaltens zur Laufzeit vorzubereiten. Daraus ergeben sich die Anforderungen an das Backend aus dem Aufbau der Laufzeitumgebung. Je nach Anforderung und Zielsetzung können Laufzeitumgebungen grundverschieden konzipiert sein. Spezielle Modell-Klassen (siehe z.B. [37]) können mit entsprechend angepassten Laufzeitumgebungen besonders effizient gelöst werden. OpenModelica ist in der Grundkonfiguration allerdings nicht auf solche speziellen Modell-Klassen ausgerichtet, sondern unterstützt ein breites Spektrum unterschiedlicher Modelle. Daher beruht die Laufzeitumgebung auf dem ODE-Konzept und ist vergleichbar mit dem überwiegenden Teil der Modelica-Compiler. Abbildung 4.1 veranschaulicht den typischen Ablauf der Laufzeitberechnung.

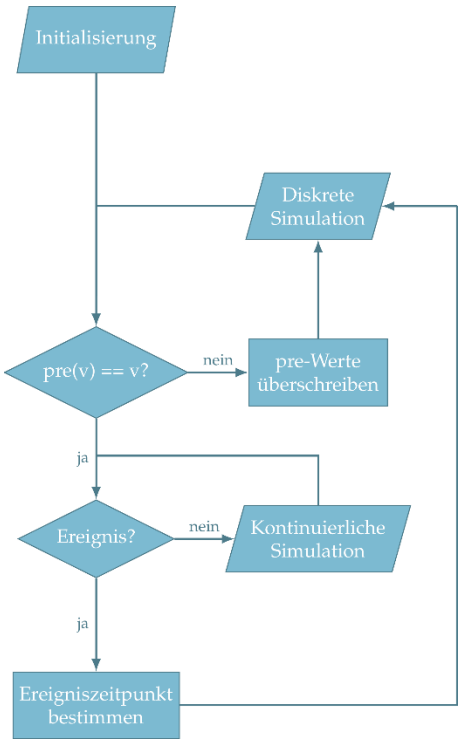

Abbildung 4.1: Zusammenspiel von Initialisierung, kontinuierlicher Simulation und diskreter Simulation zur Berechnung des dynamischen Verhaltens hybrider Systeme.

Zu Beginn der Simulation wird ein Gleichungssystem für die Initialisierung gelöst. Anschließend wird überprüft, ob konsistente Startbedingungen vorliegen (siehe Abbildung 4.1, `pre(v)==v?`). Ist dies nicht der Fall, werden alle pre-Werte, dies sind die linken Grenzwerte, mit den rechten Grenzwerten überschrieben und anschließend das System für die diskrete Simulation gelöst. Danach wird erneut auf konsistente Startbedingungen geprüft und ggf. der zuvor beschriebene Prozess wiederholt, bis konsistente Startbedingungen vorliegen. Nun wird geprüft, ob ein Ereignis vorliegt, oder seit dem letzten Auswertungsschritt eingetreten ist. Ist dies nicht der Fall, wird das System für die kontinuierliche Simulation gelöst

und anschließend wird erneut auf Ereignisse untersucht. Dieser Vorgang wiederholt sich, bis ein Ereignis eintritt oder die Simulation beendet wird. Ist ein Ereignis detektiert worden, wird zunächst der Ereigniszeitpunkt genauer bestimmt, der zwischen dem Zeitpunkt des vorherigen und des aktuellen Auswertungsschritts liegt. Anschließend wird die diskrete Simulation für den Ereigniszeitpunkt ausgeführt. Danach wird erneut auf konsistente Startbedingungen geprüft, und der Vorgang knüpft an den bereits zuvor beschriebenen Ablauf an.

Dieser Prozess enthält drei unterschiedliche Gleichungssysteme, die zur Laufzeit gelöst werden müssen und hierfür in dem Backend aufbereitet werden. Dies sind die Systeme für die *Initialisierung*, *kontinuierliche Simulation* und *diskrete Simulation*. Die Variablen (siehe Tabelle 4.1) und Gleichungen (siehe Gleichungen (4.3)-(4.5)) dieser Systeme sind gemeinsam im flachen Modell beschrieben und werden im Backend den entsprechenden Systemen zugeordnet.

Tabelle 4.1: Übersicht über die verwendete Symbolik der Variablen zur Beschreibung von Modelica-Modellen.

Symbol	Beschreibung
$\boldsymbol{x(t)}$	Zustandsvektor
$\boldsymbol{\dot{x}(t)}$	Ableitung des Zustandsvektors
$\boldsymbol{y(t)}$	Vektor der algebraischen Variablen
$\boldsymbol{d(t)}$	Diskrete Variablen (rechter Grenzwert)
$\boldsymbol{d^{pre}(t)}$	Linker Grenzwert der diskreten Variablen
$\boldsymbol{p}$	Parametervektor
$\boldsymbol{t}$	Simulationszeit

Die unterschiedlichen Systeme für die Initialisierung, kontinuierliche Simulation und diskrete Simulation unterscheiden sich vornehmlich durch die jeweils zu berechnenden Unbekannten. Tabelle 4.2 beschreibt, um welche es sich im jeweiligen Phase handelt.

Tabelle 4.2: Diese Tabelle beschreibt, welche Variablen in den jeweiligen Phasen als bekannt bzw. unbekannt angesehen werden. Am linken Rand der Tabelle sind die Kurzschreibweisen $\mathbf{z}(t)$ und $\boldsymbol{\omega}(t)$ gekennzeichnet.

	Symbol	Initialisierung	Diskrete Simulation	Kontinuierliche Simulation
$\boldsymbol{z(t)}$	$\dot{x}(t)$	unbekannt	unbekannt	unbekannt
	$y(t)$	unbekannt	unbekannt	unbekannt
	$d(t)$	unbekannt	unbekannt	bekannt
$\boldsymbol{\omega(t)}$	$x(t)$	unbekannt	bekannt	bekannt
	$d^{pre}(t)$	unbekannt	bekannt	bekannt
	p	unbekannt	bekannt	bekannt
	t	bekannt	bekannt	bekannt

Für eine kompaktere Schreibweise werden die Unbekannten der diskreten Simulation zu $\boldsymbol{z}(t)$ und die für die Initialisierung noch zusätzlichen Unbekannten zu $\boldsymbol{\omega}(t)$ zusammengefasst.

$$\boldsymbol{z}(t) \coloneqq (\dot{\boldsymbol{x}}(t) \quad \boldsymbol{y}(t) \quad \boldsymbol{d}(t))^{\top} \tag{4.1}$$
$$\boldsymbol{\omega}(t) \coloneqq (\boldsymbol{x}(t) \quad \boldsymbol{d}^{pre}(t) \quad \boldsymbol{p})^{\top} \tag{4.2}$$

Entsprechend können die Gleichungen eines Modelica-Modells in die drei verschiedenen Kategorien der kontinuierlichen Simulation (4.3), diskreten Simulation (4.4) und Initialisierung (4.5) aufgeteilt werden:

$$\boldsymbol{0} = \boldsymbol{f}(\dot{\boldsymbol{x}}(t), \boldsymbol{y}(t), \boldsymbol{d}(t), \boldsymbol{x}(t), \boldsymbol{d}^{pre}(t), \boldsymbol{p}, t) \tag{4.3}$$
$$\boldsymbol{0} = \boldsymbol{g}(\dot{\boldsymbol{x}}(t), \boldsymbol{y}(t), \boldsymbol{d}(t), \boldsymbol{x}(t), \boldsymbol{d}^{pre}(t), \boldsymbol{p}, t) \tag{4.4}$$
$$\boldsymbol{0} = \boldsymbol{h}(\dot{\boldsymbol{x}}(t), \boldsymbol{y}(t), \boldsymbol{d}(t), \boldsymbol{x}(t), \boldsymbol{d}^{pre}(t), \boldsymbol{p}, t) \tag{4.5}$$

Das Modellverhalten wird von allen dieser Gleichungen bestimmt und setzt sich aus den Simulationsgleichungen (4.3) und (4.4) sowie den initialen Bedingungen (4.5) zusammen. Das dynamische Modellverhalten wird durch das DAE-System (4.3) beschrieben. Dieses wird durch die Indexreduktion in ein System gewöhnlicher Differentialgleichungen mit zusätzlichen algebraischen Gleichungen transformiert:

$$\begin{pmatrix} \dot{\boldsymbol{x}}(t) \\ \boldsymbol{y}(t) \end{pmatrix} = \tilde{\boldsymbol{f}}(\boldsymbol{x}(t), \boldsymbol{d}(t), \boldsymbol{d}^{pre}(t), \boldsymbol{p}, t) \tag{4.6}$$

Mit diesem System werden während der kontinuierlichen Simulation für einen gegebenen Zeitpunkt t_i die Ableitungen der Zustände $\dot{\boldsymbol{x}}(t)$ ausgewertet. Die rechte Seite von Gleichung (4.6) hängt hierbei lediglich von bekannten Größen ab (vgl. Tabelle 4.2). Auf Basis von $\dot{\boldsymbol{x}}(t)$ werden die Zustände des dynamischen Systems durch die folgende Integration für den nächsten Zeitpunkt $t_i + \Delta t$ bestimmt:

$$\boldsymbol{x}(t_i + \Delta t) = \boldsymbol{x}(t_i) + \int_{t_i}^{t_i + \Delta t} \dot{\boldsymbol{x}}(t)\, dt \tag{4.7}$$

Während der Ereignisbehandlung muss das System der diskreten Simulation für den Zeitpunkt des Ereignisses t_e gelöst werden. Dieses besteht aus den dynamischen Gleichungen und zusätzlich aus den diskreten Gleichungen. Somit ergibt sich das folgende System, das nach $\boldsymbol{z}(t_e)$ gelöst wird:

$$\begin{aligned}\begin{pmatrix}\dot{\boldsymbol{x}}(t_e)\\ \boldsymbol{y}(t_e)\end{pmatrix} &= \tilde{\boldsymbol{f}}(\boldsymbol{x}(t_e), \boldsymbol{d}(t_e), \boldsymbol{d}^{pre}(t_e), \boldsymbol{p}, t)\\ \boldsymbol{0} &= \boldsymbol{g}(\dot{\boldsymbol{x}}(t_e), \boldsymbol{y}(t_e), \boldsymbol{d}(t_e), \boldsymbol{x}(t_e), \boldsymbol{d}^{pre}(t_e), \boldsymbol{p}, t)\end{aligned} \tag{4.8}$$

Die rechte Seite dieses Systems ist nicht vollständig bekannt, weshalb im Vergleich zur kontinuierlichen Simulation unterschiedliche algebraische Schleifen entstehen können. Diese algebraischen Schleifen können nun auch zu gemischten Gleichungssystem führen, also zu solchen, bei denen sowohl kontinuierliche als auch diskrete Iterationsvariablen vorkommen. Dies erschwert den Lösungsprozess und wird daher in vielen Fällen nicht unterstützt.

Sowohl die kontinuierliche als auch die diskrete Simulation hängt von den Größen $\boldsymbol{\omega}(t)$ ab. Diese müssen daher vor dem ersten Auswerten dieser Systeme bestimmt werden. Hierfür kommen die initialen Bedingungen (4.5) zum Einsatz. Mit ihnen wird folgendes algebraisches Gleichungssystem aufgebaut, das zu Beginn (t_0) einmal gelöst werden muss:

$$\begin{aligned}\begin{pmatrix}\dot{\boldsymbol{x}}(t_0)\\ \boldsymbol{y}(t_0)\end{pmatrix} &= \tilde{\boldsymbol{f}}(\boldsymbol{x}(t_0), \boldsymbol{d}(t_0), \boldsymbol{d}^{pre}(t_0), \boldsymbol{p}, t_0)\\ \boldsymbol{0} &= \boldsymbol{g}(\dot{\boldsymbol{x}}(t_0), \boldsymbol{y}(t_0), \boldsymbol{d}(t_0), \boldsymbol{x}(t_0), \boldsymbol{d}^{pre}(t_0), \boldsymbol{p}, t_0)\\ \boldsymbol{0} &= \boldsymbol{h}(\dot{\boldsymbol{x}}(t_0), \boldsymbol{y}(t_0), \boldsymbol{d}(t_0), \boldsymbol{x}(t_0), \boldsymbol{d}^{pre}(t_0), \boldsymbol{p}, t_0)\end{aligned} \tag{4.9}$$

Bei dieser Berechnung sind alle Variablen, mit Ausnahme des Initialisierungszeitpunktes t_0, unbekannt.

Ursprünglich wurde in dem OpenModelica-Backend lediglich ein Gleichungssystem für die Berechnung aller drei beschriebenen Phasen aufbereitet. Dieses System entsprach dem der diskreten Simulation mit $\boldsymbol{z}(t)$ als Unbekannte. Zum Auswerten des Systems für die kontinuierliche Simulation wurden die diskreten Variablen $\boldsymbol{d}(t)$ fixiert. Das gleiche System wurde auch zur Berechnung der Initialisierung genutzt. Hierbei wurde das System als Nebenbedingung eines Optimierungsproblems verwendet [38], für das aus den initialen Gleichungen eine Zielfunktion aufgebaut wurde, die ein globales Minimum von 0 besitzt. Dieses Minimum wird genau dann angenommen, wenn sich das System in einem gültigen initialen Zustand befindet. Dieser Ansatz ist für hybride Systeme problematisch, da durch die diskreten Variablen die Zielfunktion keinen stetigen Verlauf mehr aufweist. Auch bei rein kontinuierlichen Systemen stößt dieser Ansatz schnell an seine Grenzen, da das Optimierungsverfahren in der Regel nur lokale Optima findet, die keiner Lösung des Initialisierungsproblems entsprechen.

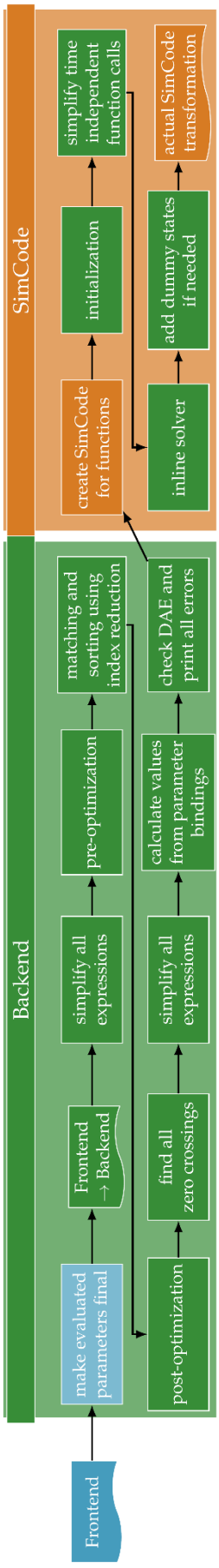

Abbildung 4.2: Die Struktur des OpenModelica-Backend im September 2013.

Aufgrund dieser Erkenntnisse wurde ein neues Verfahren zur Lösung des Initialisierungsproblems auf Basis eines symbolischen Ansatzes entwickelt. Dieser Ansatz baute auf dem damaligen Backend auf, das bis hierhin lediglich auf die Aufbereitung eines einzigen Gleichungssystems zugeschnitten war. Nachdem die gesamten Transformationsschritte für dieses System abgeschlossen waren, wurde es dupliziert und die Kopie wurde speziell für die Initialisierung aufbereitet. Als Resultat erhielt man zwei Gleichungssysteme: eines für die (kontinuierliche und diskrete) Simulation und eines für die Initialisierung.

In den Grundlagen wurde bereits auf den Aufbau des OpenModelica-Compilers eingegangen. Abbildung 4.2 beschreibt den detaillierten Aufbau des Backend in der Ausgangsversion von dieser Arbeit vom September 2013. In der Darstellung ist das Frontend blau, das Backend grün und SimCode (als Vorstufe der Codegenerierung) orange dargestellt. Auffällig ist, dass diese Phasen nicht strikt getrennt sind und z.B. essentielle Aufgaben des Backend in der SimCode-Phase durchgeführt wurden. Dies resultierte aus der zuvor beschriebenen Entwicklungsgeschichte des Initialisierungsansatzes.

Das entscheidende Problem der damaligen Backend-Struktur liegt darin, dass das System für die Simulation auch die Informationen enthielt, die später für die Initialisierung benötigt wurden, wie z.B. den `homotopy`-Operator. Dies schränkte das Optimierungspotential für die Simulationsgleichungen ein. Daher ist es notwendig, die unterschiedlichen Systeme möglichst früh aufzutrennen und individuell zu optimieren.

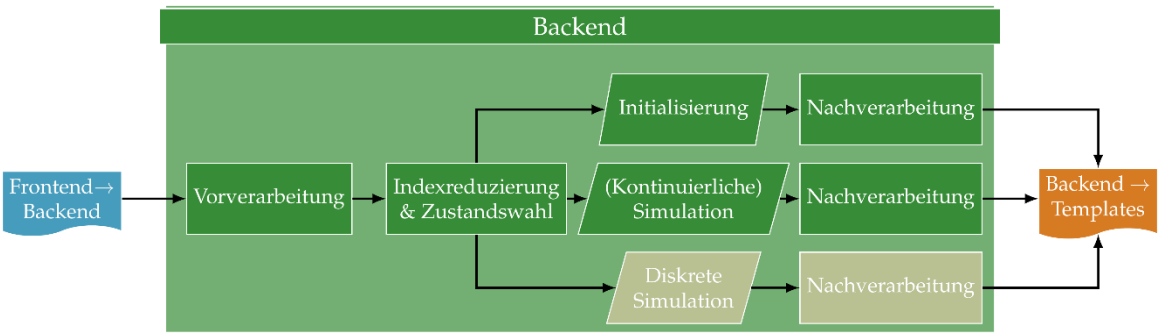

Abbildung 4.3: Grundidee des neuen Backend-Designs. Die hell dargestellte Phase für die diskrete Simulation wurde bisher nicht umgesetzt.

Die obige Abbildung zeigt, wie die neue Backend-Struktur konzeptionell darstellt. Zunächst findet eine Vorverarbeitung des flachen Modells statt, ähnlich des vorherigen Aufbaus. Hier werden Optimierungen durchgeführt, die für alle Systeme relevant sind. Danach folgt die Kausalisierung inklusive Index-Reduktion und Zustandswahl. In diesem Prozess wird das DAE zu einem System mit differentiellem Index 1 transformiert, falls dieser Index zuvor größer war. Dies lässt sich anschaulich als das Aufbrechen von Zwangsbedingungen (algebraischen Abhängigkeiten) zwischen Zuständen begreifen. Es werden auch die tatsächlichen Zustände festgelegt und gegebenenfalls zusätzliche Gleichungen abgeleitet. Diese Informationen müssen in allen Systemen nach der Aufspaltung konsistent sein, weshalb die Aufspaltung erst nach diesem Schritt erfolgt. Jedes der speziellen Systeme für die Initialisierung, diskrete Simulation und kontinuierliche Simulation wird anschließend durch eine angepasste Liste von

Optimierungsmodulen optimiert. Anhang A enthält eine Aufstellung mit Kurzbeschreibung aller verwendeten Optimierungsmodule der Vor- und Nachverarbeitung.

Die nachfolgenden Abschnitte gehen genauer auf die einzelnen Phasen des neuen Backend-Konzepts ein.

4.1 Umsetzung der Vorverarbeitung des flachen Modells

Die Vorverarbeitung des flachen Modells existierte bereits in dem alten Backend-Konzept. Es besteht aus einer Liste von Modulen, die auf dem flachen Modell angewendet werden und dieses für die Kausalisierung vorbereiten. Für das neue Backend-Design wurden die Module überarbeitet und neu zusammengestellt. So musste beispielweise das Modul `encapsulateWhenConditions` aus der Nachverarbeitung in die Vorverarbeitung verschoben werden, da es neue Variablen einführt, die sowohl in der Simulation als auch in der Initialisierung verwendet werden, und daher konsistent definiert sein müssen. Die Abbildung 4.4 veranschaulicht den Ablauf der Vorverarbeitung und der Anhang A enthält eine Kurzbeschreibung aller Module.

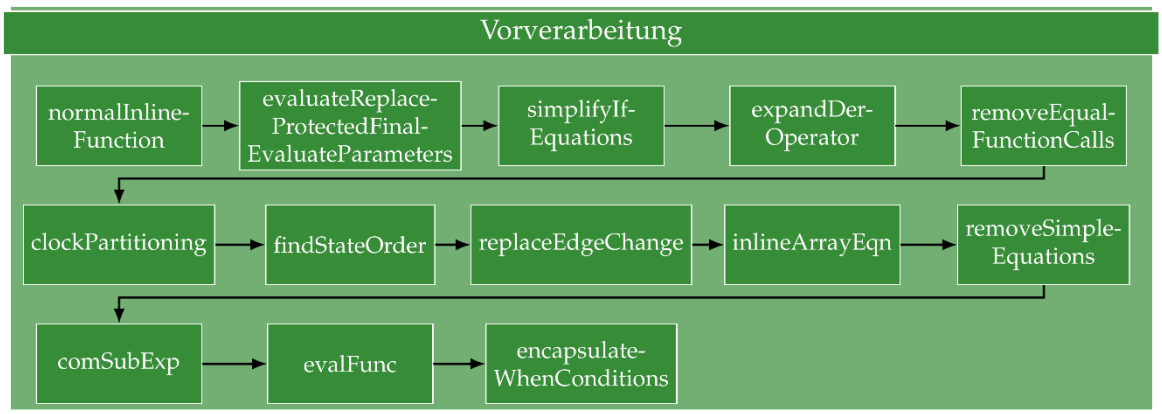

Abbildung 4.4: Die Vorverarbeitung des flachen Modells als Vorbereitung auf die Kausalisierung.

Das erste Modul der Vorverarbeitung heißt `normalInlineFunction`. Dieses Modul ersetzt Funktionsaufrufe, markiert mit Inline-Annotationen, durch ihren Funktionskörper. Anschließend wird das Modul `evaluateReplaceProtectedFinalEvaluateParameters` ausgeführt, welches die Parameter auswertet, die zur Laufzeit nicht verändert werden können. Dies sind im Wesentlichen strukturelle Parameter und solche die als `protected` definiert sind. Durch das Auswerten von Parametern, können in einigen Fällen if-Gleichungen und Ausdrücke vereinfacht bzw. eliminiert werden. Dies geschieht in dem Modul `simplifyIfEquations`. Das Modul `expandDerOperator` berechnet die Ableitung nach der Zeit für alle Ausdrücke, die innerhalb des `der`-Operators stehen. Das Modul `removeEqualFunctionCalls` versucht durch Substitution mehrfach auftretender Funktionsaufrufe das Gleichungssystem für die Weiterverarbeitung zu optimieren. Das Modul `clockPartitioning` unterteilt das flache Modell in unabhängige Gleichungssysteme. Dies führt in den nachfolgenden Modulen zu Performance-Verbesserungen, sofern deren Laufzeit schlechter als linear zur Systemgröße skaliert. Zusätzlich bestimmt das Modul die clock-Partitionen für die mit Modelica 3.3 eingeführten *Synchronous Features*. Das Modul

`findStateOrder` sammelt Informationen über die Zusammenhänge von potentiellen Zuständen und ihren Ableitungen für die Indexreduktion. In dem Modul `replaceEdgeChange` werden die Standardfunktionen `edge` und `change` durch äquivalente Ausdrücke ersetzt, wodurch die Anzahl der im weiteren Prozess zu berücksichtigen Fälle reduziert wird. Das Modul `inlineArrayEqn` flacht alle Array-Gleichungen aus. Das bedeutet, dass für jedes Element eine eigene skalare Gleichung angelegt wird. Das Modul `removeSimpleEquations` detektiert Alias-Variablen und entfernt diese aus dem Gleichungssystem. Das Modul `comSubExp` spürt Teilausdrücke auf, die in mehreren Gleichungen vorkommen und entfernt diese, falls möglich, durch Gauß-Elimination. Das Modul `evalFunc` wertet konstante Funktionensaufrufe vollständig oder partiell aus, wobei einzelne konstante Outputs als neue Gleichungen eingeführt werden. Das Modul `encapsulateWhenConditions` führt für jede `when`-Bedingung eine bool'sche Variable ein.

Darüber hinaus gibt es eine Reihe optionaler Module, die vom Benutzer manuell aktiviert werden können (siehe [9]). Beispielweise wurde das optionale Modul `unitChecking` speziell für die Arbeit mit großen biologischen Netzen entwickelt, ist allerdings nicht auf diese beschränkt. Es überprüft das Gleichungssystem anhand von physikalischen Einheiten auf Plausibilität. Dies wird im folgenden Abschnitt näher erläutert.

4.1.1 Einheitenkontrolle

Modellvariablen besitzen in Modelica eine Reihe von Attributen (siehe [5, p. 50f]). Variablen, die sich von `type Real` ableiten, besitzen unter anderem das Attribut `unit`, in dem eine physikalische Einheit gespeichert werden kann (siehe [5, p. 245f]). Diese zusätzliche Information wird für die Darstellung und Konvertierung der Simulationsergebnisse genutzt und hilft somit bei der Interpretation der Ergebnisse.

Darüber hinaus können die Einheiten-Informationen verwendet werden, um Gleichungen auf strukturelle Unstimmigkeiten hin zu untersuchen. Dadurch können dem Modellierer gezielt Informationen über Modellierungsfehler geliefert werden.

Aufgabe des Modules `unitChecking` ist es, auf Basis der angegebenen physikalischen Einheiten, möglichst vielen Variablen plausible physikalische Einheiten zuzuordnen. Hierfür wird das Gleichungssystem nach direkten und indirekten Zusammenhängen analysiert. So haben Alias-Variablen ($a = b,\ a \pm b = 0$) naturgemäß die gleiche physikalische Einheit. Sofern für eine der beiden Variablen keine Einheit spezifiziert ist, wird ihr automatisch die Information der anderen Variable zugewiesen. Neben diesen direkten Zusammenhängen können auch Einheiten-Informationen über indirekte Zusammenhänge weitergegeben werden, wie das Beispiel $c \cdot (d + e) = f$ veranschaulicht. Wenn zumindest für c oder f und d oder e Einheiten-Informationen vorliegen, können daraus die Einheiten der übrigen Variablen abgeleitet werden.

Eine weitere, für die Modellierung weitaus wichtigere, Aufgabe des Modules besteht in der Plausibilitätsprüfung des Gleichungssystems. Häufig geschehen Eingabefehler beim Übertragen von Gleichungen, es werden Terme falsch geklammert oder ein falscher

Rechenoperator taucht in den Gleichungen auf. Solche Fehler sind manuell schwer aufzuspüren und kosten daher viel Zeit. Sofern genügend Modellvariablen und –parametern Einheiten-Informationen zugewiesen wurden, können Fehlerquellen in vielen Fällen automatisch detektiert werden.

Die Funktionsweise der Einheitenkontrolle wird anhand der Gleichung (4.10) demonstriert, die dem Anwendungsfall aus Anhang B entnommen ist.

$$v_5 = v_m \frac{[F6P]^2}{k_5 \cdot \left(1 + \kappa \frac{[ATP]^2}{[ADP]^2}\right) + [F6P]^2} \tag{4.10}$$

Zunächst einmal werden die Modell-Gleichungen in eine Baum-Struktur überführt. Dies ist zur Speicherung und algorithmischen Verarbeitung von mathematischen Ausdrücken üblich. Der Baum zu Gleichung (4.10) ist in Abbildung 4.5 zu sehen.

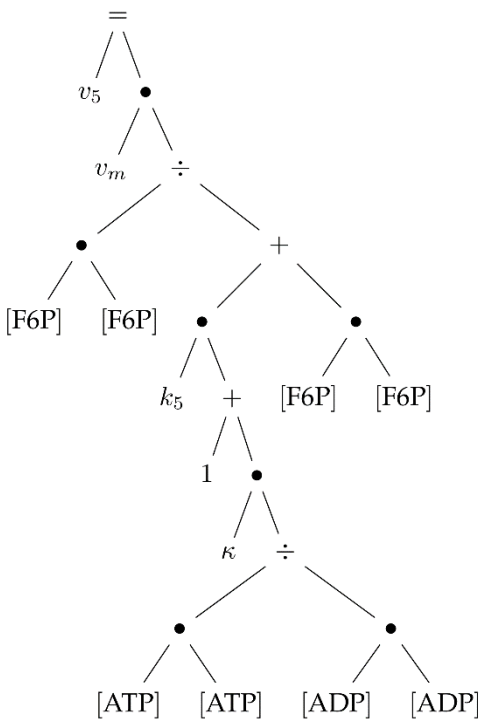

Abbildung 4.5: Baum-Struktur der Gleichung (4.10), wie sie im Compiler verwendet wird.

Die mathematischen Operationen stehen hierbei an den inneren Knoten. Dies sind z.B. Addition, Multiplikation und auch die Gleichheit, die die Wurzel des Baums bildet. In dem Beispiel aus Abbildung 4.5 handelt es sich um einen Binärbaum, doch dies muss im Allgemeinen nicht gelten. So gibt es beispielsweise das Vorzeichen als unäre Operation und Funktionsaufrufe, mit beliebig vielen Argumenten.

Neben den Gleichungen verwaltet die Einheitenkontrolle noch eine Liste aller Variablen und Parameter des Systems mit deren physikalischen Einheiten.

Der Algorithmus traversiert den Baum und versucht jedem Knoten, eine Einheit zuzuordnen. Zu Beginn wird jedem Knoten eine Einheit zugeordnet: Knoten, die eine Variable mit Einheiten-Information repräsentieren, wird diese Einheiten und allen übrigen Knoten eine unbestimmte Einheit zugeordnet. Anschließend wird der Baum in Nebenreihenfolge (bottom-up/post-order) traversiert, und an jedem Knoten wird geprüft, ob eine unbestimmte Einheit aufgrund der direkten Verbindungen bestimmt werden kann oder ob eine Inkonsistenz vorliegt. Im Falle von widersprüchlichen Einheiten werden entsprechende Meldungen an den Benutzer zurückgegeben.

4.2 Umsetzung der Simulation

Nach der Indexreduktion wird das Gleichungssystem für die unterschiedlichen Phasen aufgeteilt und unabhängig voneinander aufbereitet. Für die kontinuierliche Simulation bedeutet dies die Aufbereitung des ODE-Systems für die Auswertung zur Laufzeit und für die diskrete Simulation demzufolge die Aufbereitung des Gleichungssystems für die Ereignisbehandlung.

In der aktuellen Implementierung werden beide Aufgabenstellungen von einem gemeinsamen System gelöst. Die Nachverarbeitung umfasst hierbei die Modul-Kette aus Abbildung 4.6. Anhang A enthält eine Aufstellung aller Module und zugehörigen Kurzbeschreibungen.

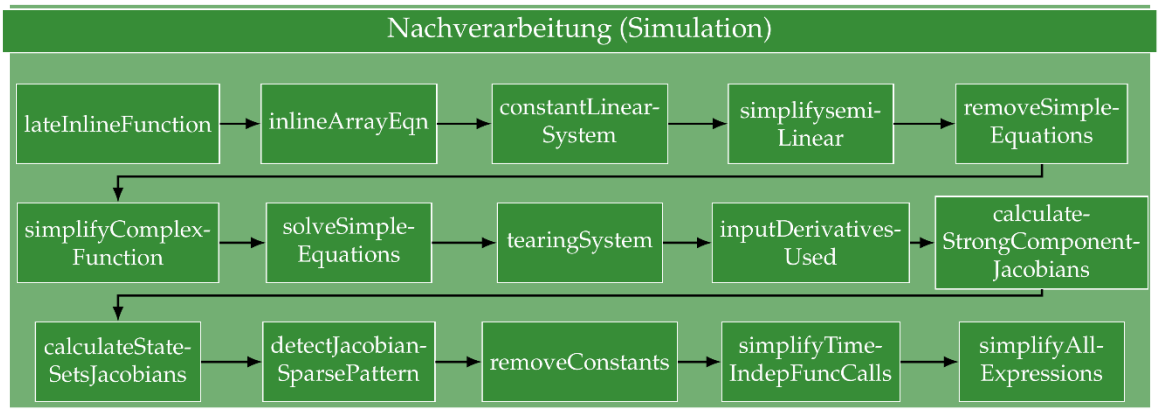

Abbildung 4.6: Nachverarbeitung des Gleichungssystems für die (kontinuierliche und diskrete) Simulation.

Die Nachverarbeitung der Simulation beginnt mit dem Modul `lateInlineFunction`, welches den Funktionsrumpf, von Funktionsaufrufen mit der Annotation `LateInline=true`, in das Gleichungssystem einbettet. Danach flacht das Modul `inlineArrayEqn` alle Array-Gleichungen aus. Das Modul `constantLinearSystem` wertet anschließend lineare Systeme der Form $A \cdot x = b$ aus, sofern A und b konstant sind. Das Modul `simplifysemiLinear` ersetzt die Modelica-Standardfunktion `semiLinear(x, positiveSlope, negativeSlope)` durch den folgenden äquivalenten Ausdruck: `if x>=0 then positiveSlope*x else negativeSlope*x`. Danach detektiert das Modul `removeSimpleEquations` einfache Gleichungen und entfernt diese aus dem Gleichungssystem. Einfache Gleichungen sind beispielweise $a \pm b = 0$, $a = \text{not } b$ und $a = 3$. Das Modul `simplifyComplexFunction` bringt Funktionsaufrufe mit mehreren

Rückgabewerten in eine Standardform, um im weiteren Übersetzungsprozess weniger Fälle berücksichtigen zu müssen. Danach versucht, `solveSimpleEquations` Gleichungen zu lösen, die zwar nur von einer Variablen abhängen, jedoch nicht trivialer Weise nach dieser aufgelöst werden können, wie zum Beispiel $\sin(x) = x + 0{,}1$. Sollte das Modul nicht in der Lage sein, eine solche Gleichung aufzulösen, wird sie als nichtlineares System mit der Dimension 1 markiert und zur Laufzeit iterativ gelöst. Danach reduziert das Modul `tearingSystem` die Dimension linearer und nichtlinearer Systeme, durch das Aufbrechen algebraischer Schleifen, falls möglich. Das Modul `inputDerivativesUsed` überprüft, ob Ableitungen von sogenannten Toplevel-Input-Variablen im Modell vorkommen. Dies ist nicht zulässig und führt daher zu einer Fehlermeldung und dem Abbruch des Übersetzungsprozesses. Danach werden symbolische Jacobimatrizen von dem Modul `calculateStrongComponentJacobians` für lineare und nichtlineare Systeme und von dem Modul `calculateStateSetsJacobians` für die dynamische Zustandswahl berechnet. Das Modul `detectJacobianSparsePattern` berechnet anschließend die dünnbesetzte Struktur des ODE-Systems. Das Modul `removeConstants` ersetzt alle Konstanten in dem Gleichungssystem durch ihren Zahlenwert, und das Modul `simplifyTimeIndepFuncCalls` vereinfacht die Standardfunktionsaufrufe von `pre`, `der`, `change` und `edge`, die nur von einem Parameter p abhängen. Zuletzt vereinfacht das Modul `simplifyAllExpressions` mathematische Teilausdrücke, wie z.B. $0 \cdot a \Rightarrow 0$ und $(1) \cdot (1) \Rightarrow 1$.

4.3 Umsetzung der Initialisierung

Die Aufgabe der Initialisierung ist es, konsistente Startwerte für diejenigen Variablen zu bestimmen, von denen das dynamische System abhängt. Bezogen auf kontinuierliche Modelle sind dies die Zustände $\boldsymbol{x}(t_0)$ und die freien Parameter. Bezogen auf hybride Systeme sind dies noch zusätzlich die linken Grenzwerte der diskreten Variablen $\boldsymbol{d}^{pre}(t_0)$.

Für die Berechnung stehen neben den Gleichungen der Simulation auch die initialen Gleichungen zur Verfügung. Dies sind zusätzliche Bedingungen, die im Modell definiert und lediglich für die Initialisierung gültig sind. Anders als bei den Gleichungssystemen der Simulation muss die Anzahl der initialen Gleichungen nicht mit denen der Unbekannten übereinstimmen. Dies wird in den folgenden zwei Abschnitten über die Initialisierung von unter- bzw. überbestimmten Systemen beschrieben.

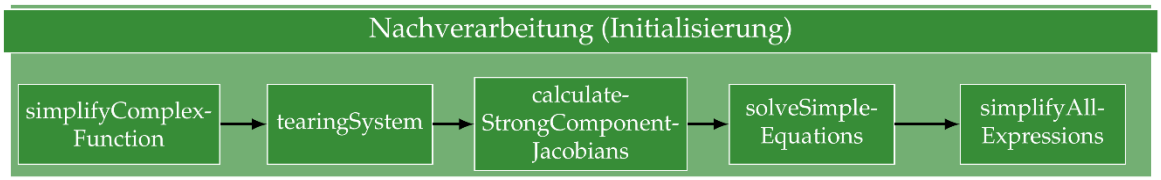

Abbildung 4.7: Die Nachverarbeitung des Gleichungssystems für die Initialisierung.

Nachdem das Gleichungssystem für die Initialisierung aufgebaut wurde, wird es durch die Nachverarbeitung (siehe Abbildung 4.3) optimiert, sodass es effizient zur Laufzeit gelöst werden kann. Die einzelnen Optimierungsmodule der Nachverarbeitung sind in Abbildung 4.7

dargestellt. Eine Liste aller Optimierungsmodule, jeweils mit Kurzbeschreibung, ist in Anhang A zu finden.

4.3.1 Initialisierung unterbestimmter Systeme

Unterbestimme Systeme beschreiben Initialisierungsprobleme, bei denen weniger initiale Bedingungen $\boldsymbol{h}(\boldsymbol{z}(t), \boldsymbol{\omega}(t), t)$ als zu initialisierende Variablen $\boldsymbol{\omega}(t_0)$ vorliegen. Durch die Analyse der Abhängigkeiten der gegebenen initialen Bedingungen $\boldsymbol{h}(\boldsymbol{z}(t), \boldsymbol{\omega}(t), t)$ von den zu initialisierenden Variablen $\boldsymbol{\omega}(t_0)$ ist es möglich, eine Menge von Variablen zu finden, die nicht über die initialen Bedingungen bestimmt werden können. Durch das Hinzufügen von Startbedingungen für diese Variablen, kann es ermöglicht werden, solche Systeme zu lösen. Dies führt im Allgemeinen nicht zu einem eindeutigen Ergebnis und sollte daher vom Modellierer vermieden werden.

Das Standard-Beispiel `BouncingBall` aus Listing 4.1 wird im Folgenden mit einer fehlenden initialen Bedingung für den Zustand `height` dazu benutzt, die Analyse der initialen Gleichungen zu veranschaulichen.

```
1  model BouncingBall
2    constant Real g = 9.81;
3    parameter Real c = 0.9;
4    parameter Real radius = 0.1;
5    Real height(start=1);
6    Real velocity(start=0, fixed=true);
7
8  equation
9    der(height) = velocity;
10   der(velocity) = -g;
11   when height <= radius then
12     reinit(velocity, -c*pre(velocity));
13   end when;
14 end BouncingBall;
```

Listing 4.1: Standard-Beispiel `BouncingBall` mit einer fehlenden initialen Bedingung für den Zustand `height`.

Dieses Beispiel enthält zwei Zustände, `height` und `velocity`, die initialisiert werden müssen. Jedoch ist nur eine initiale Bedingung für `velocity` gegeben, und daher ist das Modell unterbestimmt. Das `start`-Attribut wird im Allgemeinen als Ausgangspunkt für iterative Lösungsverfahren benutzt und beschreibt eine Näherung der Lösung. In Kombination mit dem Attribut `fixed=true` wird der Startwert zu einer impliziten initialen Gleichung für die jeweilige Variable, hier `velocity=velocity.start`. Ziel ist es nun, automatisch weitere initiale Gleichungen nach diesem Schema zu generieren, bis das System den vollen Rang hat. Daher muss zunächst analysiert werden, welche der zu initialisierenden Variablen $\boldsymbol{\omega}(t_0)$ nicht durch die gegebenen initialen Gleichungen berechnet werden können.

Die Grundidee des Algorithmus zur Bestimmung der zusätzlichen Bedingungen ist, dass das resultierende System die gleiche Anzahl an Variablen und Gleichungen besitzen muss. Somit werden zunächst abstrakte Gleichungen in den Variablen/Gleichungen-Graphen aus Abbildung 4.8 eingeführt, bis das System quadratisch ist. Diese zusätzlichen Gleichungen (hier

q_1^{eq}) werden mit allen zu initialisierenden Variablen $\boldsymbol{\omega}(t_0)$, in diesem Fall lediglich den Zuständen, verbunden:

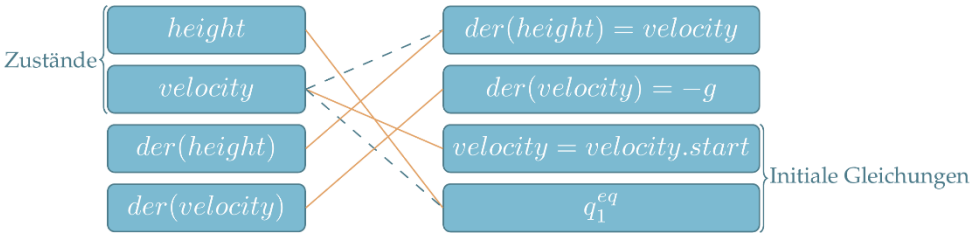

Abbildung 4.8: Ergebnis des Matching für das Beispiel `BouncingBall` *aus Listing 4.1 mit einer zusätzlichen, abstrakten, initialen Gleichung* q_1^{eq}.

Für den so erzeugten Graphen existiert ein perfektes Matching, sofern das zugrundeliegende dynamische System nicht bereits strukturell singulär ist. Ist das perfekte Matching zudem eindeutig, ist auch die Wahl der zusätzlichen initialen Gleichungen eindeutig. Dies ist in dem Beispiel aus Listing 4.1 der Fall. Im Allgemeinen ist jedoch das perfekte Matching nicht eindeutig. Dies bedeutet, dass es starke Zusammenhangskomponenten gibt, innerhalb derer die Wahl der initialen Gleichung durch eine Heuristik getroffen werden kann. Denkbar ist es Variablen mit Startwerten zu bevorzugen.

Algorithmus 4.1: Initialisierung unterbestimmter Systeme

1. Wende einen Algorithmus zur Indexreduktion auf das dynamische System an.
2. Füge die zu initialisierenden Variablen $\boldsymbol{\omega}(t_0)$ und die initialen Bedingungen dem dynamischen System hinzu, um das Initialisierungsproblem aufzubauen.
3. Baue den zugehörigen bipartiten Graphen auf.
4. Füge so viele initiale Gleichungen q_i^{eq} zu dem System hinzu, bis dieses quadratisch ist, und verbinde jeden Knoten q_i^{eq} mit allen Knoten aus $\boldsymbol{\omega}(t_0)$.
5. Suche ein perfektes Matching.
6. Generiere aus jedem Knoten q_i^{eq} eine initiale Bedingung $x_i = x_i.start$, wobei x_i diejenige Variable ist, die q_i^{eq} in Schritt 5 zugewiesen wurde.

Der Mehraufwand von Algorithmus 4.1 im Vergleich zu dem Vorgehen bei einem System mit ausreichend vielen initialen Bedingungen ist lediglich Schritt 4 und 6. Der Aufwand beider Schritte verhält sich linear zur Anzahl der fehlenden initialen Bedingungen. Die Anzahl der fehlenden initialen Bedingungen ist grundsätzlich unabhängig von der Anzahl n der Gleichungen. Somit ergibt sich ein konstanter Mehraufwand, der allerdings in Anbetracht der Schritte 1 (Indexreduktion) und 5 (Matching) vernachlässigt werden kann (siehe. [10, p. 54f]). Somit bestimmt der Aufwand des Matching-Algorithmus die Komplexität von Algorithmus 4.1, die mit $\mathcal{O}(\tau \cdot n)$ abgeschätzt werden kann (vgl. [39]). τ steht für die Anzahl der Kanten im Variablen/Gleichungen-Graphen und somit gilt $\tau \geq n$. In vielen praxisrelevanten

physikalischen Modellen kann τ allerdings durch eine Konstante abgeschätzt werden, die unabhängig von n ist (siehe [10, p. 35]), wodurch sich eine lineare Komplexitäts-Abschätzung in Abhängigkeit zu der Anzahl der Modellgleichungen ergibt.

Sollte kein perfektes Matching existieren und somit das System singulär sein, bleibt der Algorithmus in Schritt 5 stecken. Hierfür gibt es zwei mögliche Ursachen: Entweder ist das zugrundeliegende dynamische System bereits singulär, oder aber es liegt ein gemischt-bestimmtes Initialisierungsproblem vor. Hierauf wird in dem entsprechenden Abschnitt 4.3.3 näher eingegangen.

4.3.2 Initialisierung überbestimmter Systeme

Modelica-Modelle basieren auf einem objektorientierten Ansatz und bestehen somit für gewöhnlich aus unterschiedlichen Komponenten. Daher sind die initialen Bedingungen auch mitunter über diese Komponenten verteilt.

Durch das Verbinden von Komponenten, können algebraische Abhängigkeiten zwischen Zuständen eingeführt werden. Der typische Ansatz der Simulation solcher Modelle basiert auf differentialalgebraischen Index-1-Systemen. Daher wird von vielen Modelica-Programmen eine Methode zur Indexreduktion verwendet, z.B. die sogenannte *Dummy-Derivative-Methode* [13]. Diese bricht algebraische Abhängigkeiten, durch das Abwerten von bestimmten Zuständen und ihren Ableitungen zu gewöhnlichen algebraischen Variablen, auf. In diesem Prozess werden allerdings keine initialen Bedingungen berücksichtigt, wodurch es dazu kommen kann, dass anschließend mehr initiale Bedingungen als notwendig definiert sind. Diese Systeme werden überbestimmte Systeme genannt.

Ein Algorithmus zur Lösung solcher Initialisierungsprobleme wurde zuerst in [38] diskutiert. Der dort vorgestellte Algorithmus löst das Problem numerisch und überführt es dafür in ein Optimierungsproblem. Dieser Ansatz hat eine Reihe von Nachteilen, wie z.B. lokale Minima und numerische Schwellwerte. Später wurde ein anderer Ansatz vorgestellt [40], der redundante Gleichungen symbolisch detektiert. Diese Gleichungen werden anschließend automatisch entfernt, um ein quadratisches und somit lösbares System zu erhalten. Der beschriebene Algorithmus zeigt allerdings in einigen Fällen eine schlechte Performance, da er in lokale Singularitäten hineinlaufen kann, die dann über eine Backtracking-Strategie und ein modifiziertes Matching des Systems aufgelöst werden. Neben diesem Nachteil hat der Ansatz noch zwei weitere Limitierungen, die ihn unattraktiv machen: der Algorithmus funktioniert nur solange die redundanten Gleichungen nicht Teil einer algebraischen Schleife sind. Gemischt-bestimmte Systeme können zudem nicht gelöst werden.

Der nachfolgend vorgestellte Ansatz analysiert das Initialisierungsproblem analytisch und entfernt konsistente Bedingungen automatisch und zeigt inkonsistente Bedingungen durch benutzerfreundliche Benachrichtigungen auf. Dies führt die symbolische Initialisierung von OpenModelica [41], [40] weiter und ermöglicht so die Initialisierung komplexer hybrider Modelle.

Das nachstehende Beispiel aus dem Paket *ModelicaTest*, das von der Modelica Association bereitgestellt wird, wird im Weiteren zur Illustration verwendet:

```
1  model TwoMassesEquationsFullInitial
2    Real x1, v1;
3    Real x2, v2;
4    Real F1, F2;
5    parameter Real M = 1;
6    parameter Real K = 1;
7    parameter Real F0 = 1;
8
9  initial equation
10   x1 = 0;
11   v1 = 0;
12   x2 = 0;
13   v2 = 0;
14
15 equation
16   der(x1) = v1;
17   M * der(v1) = F1 + F2;
18   der(x2) = v2;
19   M * der(v2) = -F2;
20   F1 = -K * x1;
21   x1 = x2;
22 end TwoMassesEquationsFullInitial;
```

Listing 4.2: Dieses Modell beschreibt zwei verbundene starre Körper, die über eine Feder mit dem Boden verbunden sind. Es enthält initiale Bedingungen für alle potentiellen Zustände, wodurch das Initialisierungsproblem überbestimmt ist.

Das Beispiel aus Listing 4.2 enthält vier potentielle Zustände. Zwei von ihnen sind von den übrigen zwei Zuständen algebraisch abhängig, wodurch der Index des Systems angehoben wird. Daher wird während des Transformationsprozesses ein Algorithmus zur Indexreduktion angewendet, der wiederum das System in ein System mit Index 1 transformiert. Das so erzeugte System hat nur noch zwei Zustände und weiterhin die ursprünglichen vier initialen Gleichungen. Sofern die definierten initialen Bedingungen (Listing 4.2, Zeile 9-13) konsistent sind, ist es möglich, zwei von ihnen zu entfernen und das resultierende System mit den herkömmlichen Methoden zu lösen. Im Weiteren wird gezeigt, wie eine effiziente Wahl der potentiell redundanten Gleichungen aussehen kann:

Dem Gleichungssystem für die Initialisierung werden so viele Dummy-Variablen (q_i^{var}) hinzugefügt, bis es die gleiche Anzahl von Gleichungen und Variablen besitzt (siehe Abbildung 4.9). Anschließend wird ein perfektes Matching für den Gleichungen/Variablen-Graphen gesucht. Die initialen Gleichungen, die den Dummy-Variablen zugeordnet wurden, werden als potentiell redundante Gleichungen betrachtet.

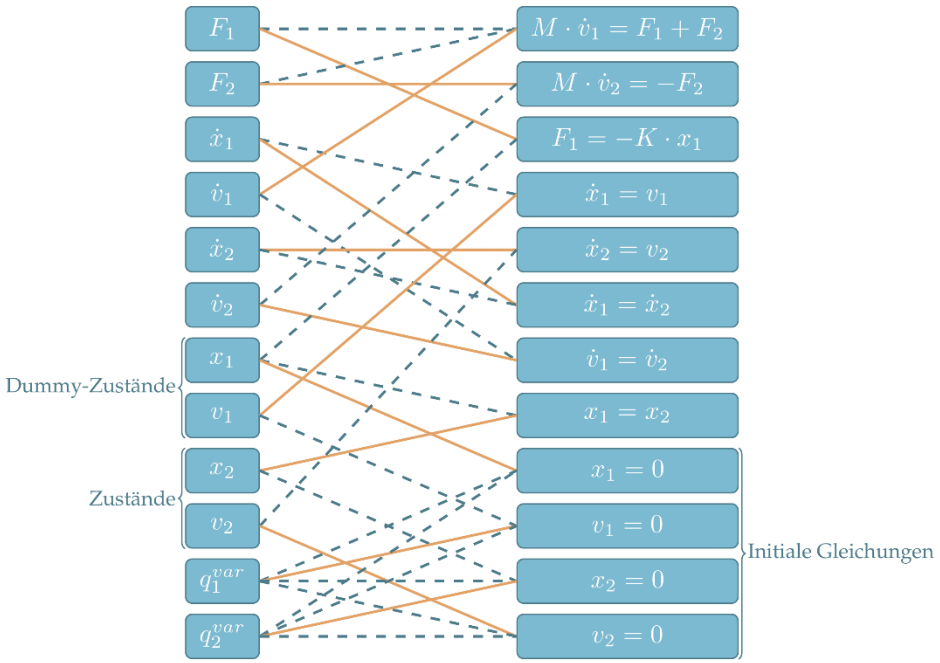

Abbildung 4.9: Das Ergebnis des Matchings für die Initialisierung des überbestimmten Beispiels ModelicaTest.OverdeterminedInitialization.Mechanical.TwoMassesEquationsFullInitial.

Die potentiell redundanten Gleichungen müssen nun auf Konsistenz geprüft werden. Dies kann grundsätzlich sowohl numerisch als auch symbolisch geschehen. Der symbolische Ansatz ist vorzuziehen, kann unter Umständen allerdings nicht ohne Weiteres während des Kompilierungsprozesses erfolgen. Sollte sich eine potentiell redundante Gleichung symbolisch zu $0 = 0$ auflösen, so ist sie konsistent und kann gefahrlos aus dem System gelöscht werden. Wird die Gleichung zu $0 = 1$, so ist das System inkonsistent (vgl. [40]). Sollte sich diese Entscheidung nicht während des Kompilierungsprozesses treffen lassen, so kann die Konsistenz numerisch bestimmt werden, nachdem das Initialisierungssystem gelöst wurde. Sollte das Residuum kleiner als ein definierter Schwellwert sein, so kann die Gleichung ebenfalls als konsistent betrachtet werden. Für diesen numerischen Abgleich ist es erforderlich, eine sinnvolle Skalierung für die Residuen zu finden. Dies ist eine noch offene Problemstellung.

Algorithmus 4.2 beschreibt die zuvor exemplarisch durchgeführte Methode allgemein.

Der Mehraufwand für die Initialisierung überbestimmter Systeme mit Algorithmus 4.2 liegt in den Schritten 4 und 6 (vgl. Algorithmus 4.1). Die Komplexität beider Schritte ist zunächst unabhängig von der Modellgröße und hängt lediglich von der Anzahl der zu viel definierten Gleichungen ab. Dahingegen muss die Komplexitätsabschätzung für die Überprüfung der Konsistenz getrennt betrachtet werden. Der numerische Konsistenz-Check verursacht keinen zusätzlichen Aufwand, wohingegen für eine verlässliche Abschätzung der symbolischen Konsistenz-Überprüfung weitere Untersuchungen notwendig sind.

Algorithmus 4.2: Initialisierung überbestimmter Systeme

1. Wende einen Algorithmus zur Indexreduktion auf das dynamische System an.
2. Füge die zu initialisierenden Variablen $\boldsymbol{\omega}(t_0)$ und die initialen Bedingungen dem dynamischen System hinzu, um so das Initialisierungsproblem aufzubauen.
3. Baue den zugehörigen bipartiten Graphen auf.
4. Füge so viele Dummy-Variablen q_i^{var} dem System hinzu, bis dieses quadratisch ist und verbinde jeden Knoten q_i^{var} mit allen initialen Gleichungen.
5. Suche ein perfektes Matching.
6. Entferne jede Dummy-Variable q_i^{var} zusammen mit der über das Matching aus Schritt 5 zugeordneten initialen Bedingung und überprüfe diese auf Konsistenz.

Sollte kein perfektes Matching existieren und somit das System singulär sein, bleibt der Algorithmus in Schritt 5 stecken. Dies kann der Fall sein, wenn es sich um ein gemischt-bestimmtes System handelt. Hierauf wird im folgenden Abschnitt näher eingegangen.

4.3.3 Initialisierung gemischt-bestimmter Systeme

In praxisrelevanten Modellen kann die Initialisierung einzelner Modellteile unterbestimmt und anderer Modellteile gleichzeitig überbestimmt sein. Somit kann es auf oberster Modellebene vermeintlich ausreichend viele initiale Bedingungen geben. Jedoch existiert kein perfektes Matching für das Initialisierungsproblem; dieses ist somit strukturell singulär. Durch Anwendung einer Kombination der zuvor beschriebenen Algorithmen für unter- und überbestimmter Systeme können diese Probleme dennoch gelöst werden. Solche Systeme werden im Weiteren gemischt-bestimmte Systeme genannt.

```
1  model MixedDetermined
2    Real x;
3    Real y;
4    Real z;
5  initial equation
6    y = 1;
7    y^2 = 1;
8  equation
9    der(x) = sin(time);
10   der(y) = sin(time);
11   z = x + y;
12 end MixedDetermined;
```

Listing 4.3: Gemischt-bestimmtes Initialisierungsproblem.

Das Beispiel aus Listing 4.3 ist ein solches gemischt-bestimmtes Modell. Es enthält zwei Zustände, x und y, und zwei initiale Gleichungen. Beide initiale Gleichungen beschreiben allerdings den Zustand y. Eine initiale Bedingung für den Zustand x fehlt. Wie bereits erwähnt, ist dieses System strukturell singulär. Die Singularität wird mit den Gleichungen/Variablen-Graphen in Abbildung 4.10 durch die beiden durchgezogenen orangen Linien veranschaulicht.

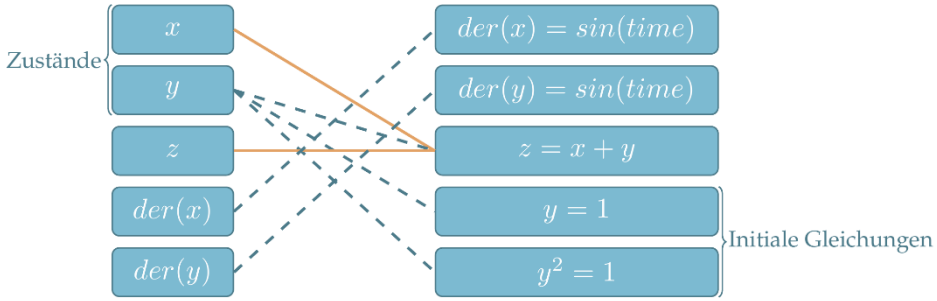

Abbildung 4.10:Strukturell singuläres gemischt-bestimmtes Initialisierungsproblem.

Damit das System lösbar wird, muss eine der initialen Gleichungen entfernt und durch eine neue Gleichung, die den Zustand x beschreibt, ersetzt werden. Der nachstehende Algorithmus beschreibt, wie dies realisiert werden kann:

Algorithmus 4.3: Initialisierung gemischt-bestimmter Systeme

1. Wende einen Algorithmus zur Indexreduktion auf das dynamische System an.
2. Füge die zu initialisierenden Variablen $\boldsymbol{\omega}(t_0)$ und die initialen Bedingungen dem dynamischen System hinzu, um das Initialisierungsproblem aufzubauen.
3. Baue den zugehörigen bipartiten Graphen auf.
4. Füge nur so viele Dummy-Variablen q_i^{var} und Dummy-Gleichungen q_i^{eq} wie nötig dem System hinzu, damit dieses quadratisch wird. Verbinde jeden Knoten q_i^{var} mit allen initialen Gleichungen und jeden Knoten q_i^{eq} mit allen Variablen aus $\boldsymbol{\omega}(t_0)$.
5. Füge jeweils eine weitere Dummy-Variable und eine weitere Dummy-Gleichung mit den zugehörigen Kanten ein.
6. Suche ein perfektes Matching. Falls dieses existiert, gehe zu Schritt 7. Falls es kein perfektes Matching gibt und sowohl die Anzahl der Dummy-Knoten q_i^{eq} kleiner als die Anzahl der zu initialisierenden Variablen $\boldsymbol{\omega}(t_0)$ ist, als auch die Anzahl der Dummy-Variablen q_i^{var} kleiner als die Anzahl der initialen Gleichungen ist, gehe zurück zu Schritt 5. Andernfalls breche ab.
7. Wandle jeden Knoten q_i^{eq} in eine initiale Bedingung $x_i = x_i.start$ um, wobei x_i diejenige Variable ist, die q_i^{eq} in Schritt 6 zugeordnet wurde.
8. Entferne jede Dummy-Variable q_i^{eq} zusammen mit der über das Matching aus Schritt 6 zugeordneten initialen Bedingung und überprüfe diese auf Konsistenz.

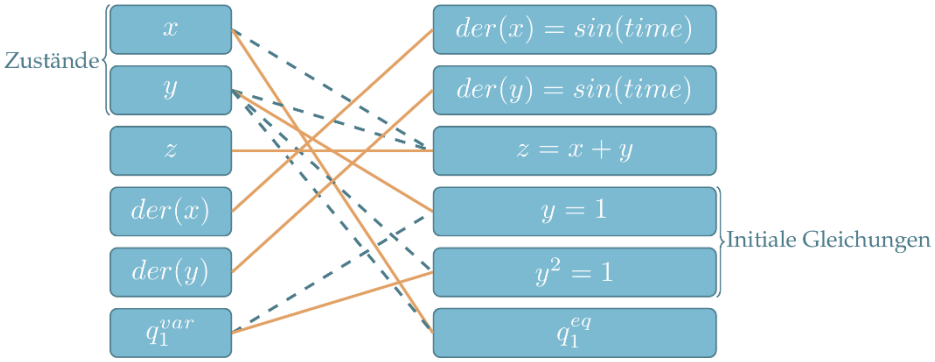

Abbildung 4.11: Gemischt-bestimmtes Initialisierungsproblem.

Angewendet auf das Beispiel aus Listing 4.3 ergibt sich das Modelica Modell aus Listing 4.4, das nun über vollständig bestimmte initiale Bedingungen verfügt.

```
1  model MixedDetermined
2    Real x;
3    Real y;
4    Real z;
5  initial equation
6    y = 1;
7    x = x.start;
8  equation
9    der(x) = sin(time);
10   der(y) = sin(time);
11   z = x + y;
12 end MixedDetermined;
```

Listing 4.4: Das Modell aus Listing 4.3 nach dem Anwenden des Algorithmus zur Lösung gemischt-bestimmter Systeme.

Der Algorithmus bricht in Schritt 6 genau dann ab, wenn das zugrundeliegende dynamische System bereits strukturell singulär ist. In diesem Fall existiert keine Lösung für das Initialisierungsproblem.

4.4 Zusammenfassung

In diesem Kapitel wurde die zu Beginn dieser Arbeit bestehende Backend-Struktur des OpenModelica-Compilers untersucht und konzeptionelle Probleme aufgezeigt. Anschließend wurde eine neue Backend-Struktur entworfen und im Rahmen des OpenModelica-Projektes umgesetzt. Hierfür wurden einige der Optimierungsmodule (siehe Anhang A) angepasst und neu zusammengestellt.

Dies ermöglicht es, komplexe hybride Systeme sowohl effizient zu simulieren, als auch zu initialisieren, da die jeweiligen Gleichungssysteme nun unabhängig voneinander optimiert werden.

Es wurde ein neues Optimierungsmodul entwickelt, das es erlaubt Gleichungen auf Basis von physikalischen Einheiten auf strukturelle Richtigkeit hin zu untersuchen. Zusätzlich weißt das Modul so vielen Variablen wie möglich plausible physikalische Einheiten zu. Dies hilft bei der

Modellierung durch das automatische Aufspüren von Modellierungsfehlern und bei der Interpretation von Ergebnissen.

Zudem wurden neue Algorithmen für die Behandlung spezieller Initialisierungsprobleme entwickelt und implementiert. Dies umfasst Lösungsstrategien für unter-, über- und gemischt-bestimmte Initialisierungsprobleme.

5 Weiterentwicklung der PNlib

Die in Abschnitt 3.2 ausführlich vorgestellte Bibliothek PNlib 1.0 wurde für die Modellierung und Simulation von zeitbasierten Petri-Netzen entwickelt. Sie erfüllt in Version 1.0 allerdings nicht den Modelica-Sprachstandard, zeigt jedoch die Möglichkeiten auf, die Modelica für die Arbeit mit Petri-Netzen bietet. Dies sind insbesondere die transparente Umsetzung des Petri-Netz-Formalismus, die freie Nutzung und Einbindung der erstellten Petri-Netze in andere Modelle und Simulationsumgebungen.

In diesem Kapitel wird beschrieben, wie die Bibliothek dahingehend erweitert wurde, dass sie Modelica-konform ist. Gleichzeitig wurde auch der Modelica-Sprachstandard von der Modelica Association weiterentwickelt und um Funktionalitäten ergänzt, die bereits in der PNlib Verwendung fanden. So wurde z.B. in der Version 3.3 des Modelica-Sprachstandards die Möglichkeit zur Definition von `impure`-Funktionen eingeführt, die in der PNlib 1.0 bereits verwendet wurden. Dadurch bedingt, musste die PNlib 1.0 auf eine spezielle Modelica-Umgebung zugeschnitten werden, die damals diese Erweiterungen zusätzlich zu dem offiziellen Sprachstandard bereitstellte. Dies war die kommerzielle Umgebung Dymola, die jedoch in ihrer aktuellen Version die PNlib 1.0 nicht mehr unterstützt.

Tabelle 5.1: Offizielle Versionen der PNlib mit den jeweils wichtigsten Änderungen.

Version	Datum	Beschreibung
1.0	04/2013	• Erste Version
1.1	01/2015	• Anpassung an Modelica 3.3 • Reproduzierbare stochastische Simulationen • Fehlerbehebung
1.2	10/2015	• Unterstützung von 64bit-Simulationen • Fehlerbehebung
1.3	03/2016	• Tokenflüsse • Testbeispiele • Fehlerbehebung
2.0	12/2016	• Kompatibel zu OpenModelica & Dymola • Neue Implementierung für Zufallszahlen • MSL 3.2.2 • Gefaltete Plätze • Fehlerbehebung

Seit Beginn der Entwicklung der Bibliothek PNlib wird angestrebt, diese mit dem quelloffene Modelica-Projekt OpenModelica zu unterstützen, um nicht nur eine freie Modellierung, sondern auch Simulation von Petri-Netzen zu ermöglichen. Neben den oben angesprochenen Problemen mit dem Modelica-Sprachstandard bot jedoch der OpenModelica-Compiler in der damaligen Version noch nicht die notwendige Stabilität und Performance.

Durch die Berücksichtigung unterschiedlicher Modelica-Programme, in erster Linie Dymola und OpenModelica, mussten die mathematischen Ausdrücke abgeändert werden, die in einem der Programme zu Problemen führte. Grund hierfür ist, dass jedes Modelica-Programm unterschiedliche symbolische Transformationen auf die mathematischen Ausdrücke anwendet. So führten z.B. Relationen der Form $^{a}/_{b} < {}^{c}/_{d}$ bei OpenModelica zu Problemen, sobald b bzw. d 0 wurden. Da die beiden Größen b und d hier Kantengewichten entsprechen und somit nicht negativ werden können, konnten diese Relationen durch die äquivalente Darstellung $a \cdot d < c \cdot b$ in der Bibliothek ersetzt werden. Mit diesen Änderungen verhält sich die Bibliothek auf den unterschiedlichen Plattformen vergleichbar.

5.1 Testumgebung

Die Arbeit an der Bibliothek hat gezeigt, dass es mitunter schwierig ist, die Auswirkungen einzelner Änderungen vorherzusehen. Zudem entwickeln sich auch die Modelica-Programme stetig weiter, wobei sich Fehler einschleichen können. In der aktuellen Dymola-Version (Dymola 2017) liefern z.B. die `and` und `or`-Operanden für Arrays in Algorithmen nicht mehr die richtigen Ergebnisse. In der Vorgängerversion funktionierte dies noch. Um die gewünschte Funktionsweise der Bibliothek sicherzustellen, ist daher intensives Testen unablässig. Mit jeder Änderung an der Bibliothek, sowie mit jeder neuen Version der Modelica-Programme, muss daher ein vollständiger Testlauf durchgeführt werden. Dies erlaubt das frühzeitige und zielgenaue Aufspüren von Fehlern und reduziert somit die Zeit, die für die Fehlersuche aufgewendet werden muss, erheblich.

Hierfür enthält die aktuelle Version der Bibliothek eine Reihe von Beispielen, die in dem Paket `PNlib.Examples` organisiert sind. Die Testbeispiele unterteilen sich in vier Unterpakete: `ConTest`, `DisTest`, `ExtTest`, `HybTest`. Die Beispiele sind so angelegt, dass sie möglichst klein und übersichtlich sind und gezielt einzelne Aspekte des xHPN-Formalismus abdecken. Zudem gibt es noch das Unterpaket `Models`, das komplexere Beispiele zu konkreten Anwendungsszenarien enthält.

Das Paket `ConTest` enthält zehn Tests für rein kontinuierliche Petri-Netze. Das Paket `DisTest` enthält dreizehn Tests für rein diskrete Testfälle. Das Paket `ExtTest` enthält dreizehn Beispiele für erweiterte Petri-Netze, also solche mit speziellen Kanten (z.B. Hemmkanten) und stochastischen Transitionen. Das Paket `HybTest` besteht aus sechzehn Tests für hybride Petri-Netze, die sowohl aus kontinuierlichen wie auch diskreten Komponenten bestehen. Die nachfolgende Grafik zeigt exemplarisch ein Beispiel aus jedem dieser Testpakete:

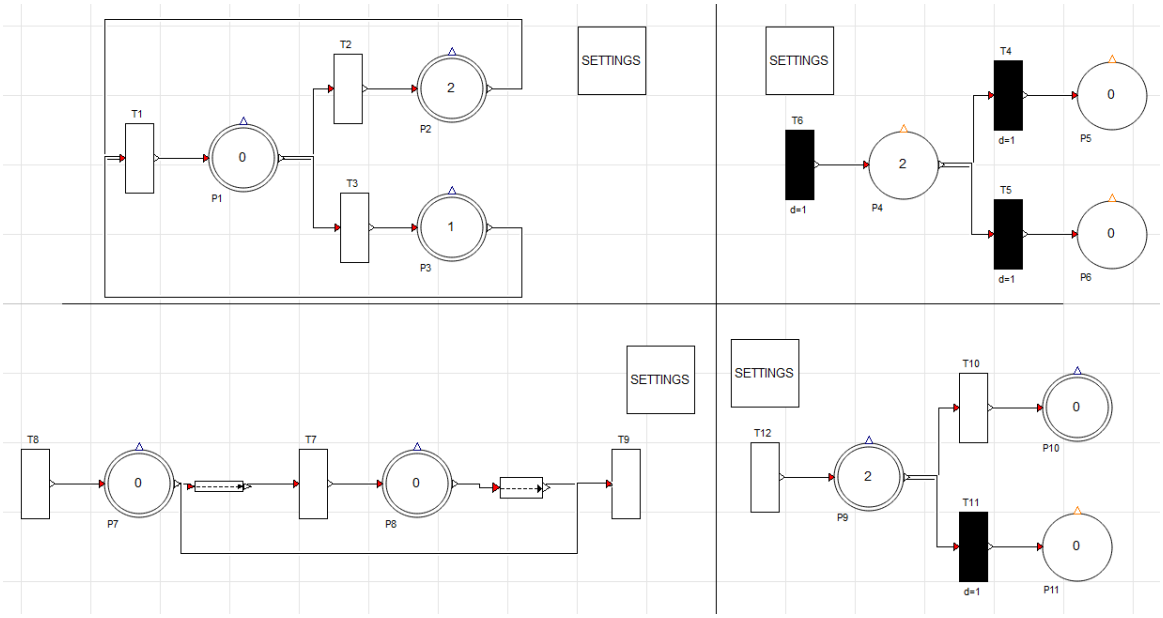

Abbildung 5.1: Ausgewählte Beispiel aus der PNlib. Oben links: `ConTest.ConflictLoop`*. Oben rechts:* `DisTest.ConflictPrio`*. Unten links:* `ExtTest.TATest`*. Unten rechts:* `HybTest.ConflictType3`*.*

Seit März 2015 ist die PNlib Bestandteil der OpenModelica-Testumgebung, die einmal am Tag automatisch ausgeführt wird. Die Ergebnisse werden in dem Diagramm aus Abbildung 5.2 zusammengefasst, welches die Gesamtanzahl der Tests (*Target*) zeigt, sowie die Anzahl der erfolgreich kompilierten (*Compile*) und simulierten Tests (*Simulate*). Zudem wird die Anzahl der Tests angezeigt, bei denen die Simulationsergebnisse mit den Musterlösungen aus der Bibliothek übereinstimmen (*Verified*).

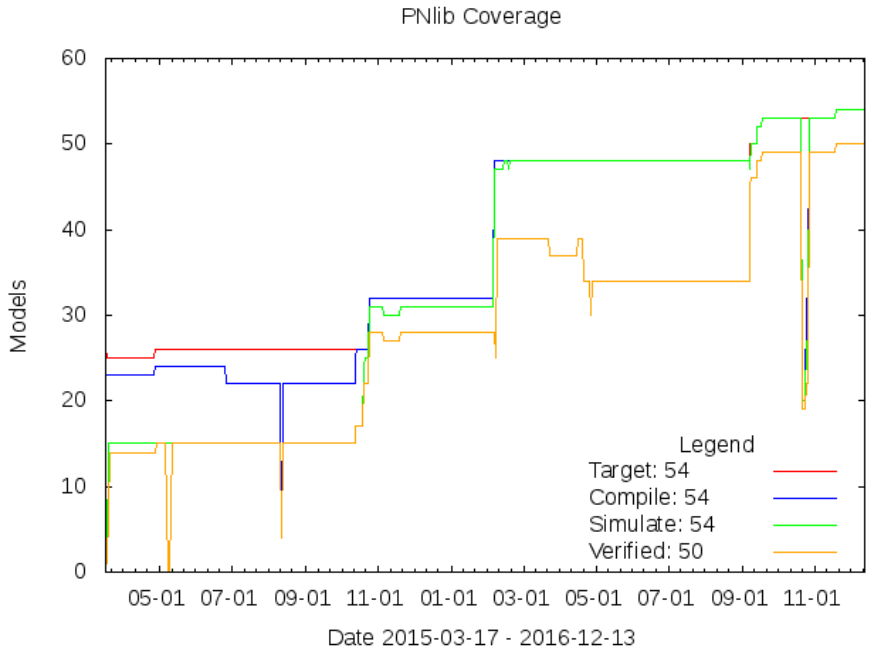

Abbildung 5.2: Verlauf des PNlib-Coverage-Tests von OpenModelica. Aktuell enthält die PNlib 54 Modelle, die getestet werden. Für vier der Modelle sind in der PNlib keine Musterlösungen definiert, weshalb bei „Verified“ lediglich 50 angezeigt wird. Somit werden aktuell alle definierten Beispiele richtig simuliert.

Das Diagramm aus Abbildung 5.2 zeigt die Entwicklung der PNlib und die Unterstützung dieser von OpenModelica. Anhand der Kurve *Target* ist zu sehen, dass der Bibliothek im Lauf der letzten Monate eine signifikante Anzahl an Testbeispielen hinzugefügt wurde. Diese sollen sicherstellen, dass im Entwicklungsprozess keine Fehler hinzukommen. Zwischendurch sind immer wieder Ausschläge nach unten zu sehen. Diese rühren entweder von Fehlern in der Bibliothek oder in dem OpenModelica-Projekt her. Da die Bibliothek täglich getestet wird, kann schnell auf solche Ausreißer reagiert werden. Derzeit werden alle Beispiele fehlerfrei ausgeführt, wie der Coverage-Test in Abbildung 5.2 zeigt.

5.2 Tokenflüsse

Bisher wurde bei den Petri-Netzen das Hauptaugenmerk auf die Knoten und ihre Zustände gelegt. Insbesondere in Netzen mit Schleifen kann es jedoch wichtig sein, zusätzlich noch den Durchfluss durch einen Knoten zu berücksichtigen, und den Fluss über eine Kante zu betrachten. Dies wird an dem Beispiel aus Abbildung 5.3 deutlich, bei dem die Markierungen der Plätze p_1, p_2 und p_3 während der Simulation konstant sind. Dennoch „wandern“ Tokens durchs Netzwerk. Dies kann erst durch die Betrachtung der Tokenflüsse nachvollzogen werden. In komplexen Anwendungen, die beispielsweise den Citrat-Zyklus oder Calvin-Zyklus enthalten, ist dies ein wichtiges Werkzeug, um den Prozess nachzuvollziehen. Siehe hierzu auch Abbildung 6.3 zu dem entsprechenden Anwendungsfall aus Kapitel 6.

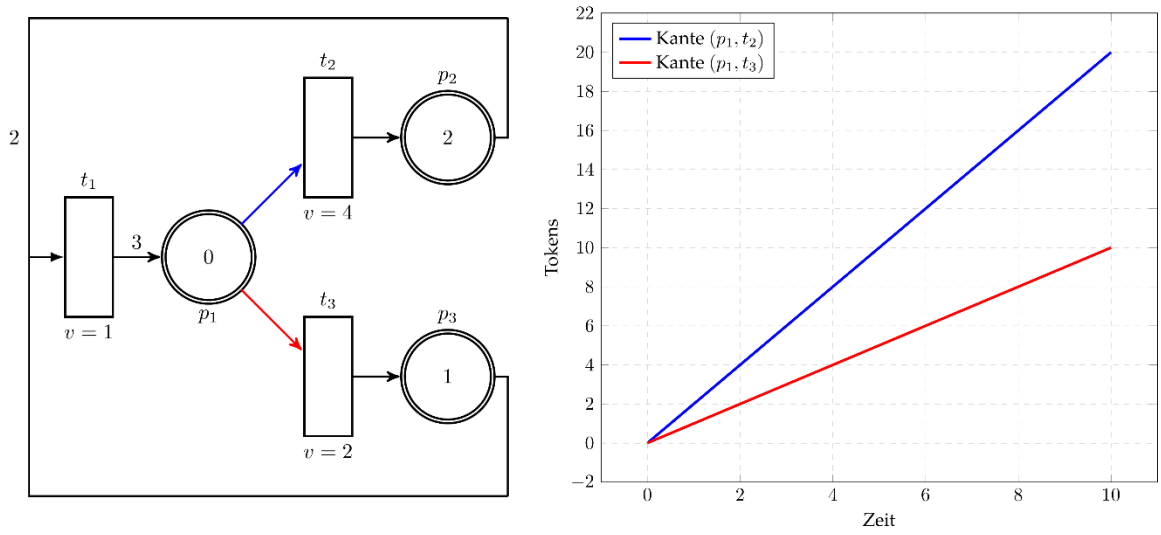

Abbildung 5.3: Das Petri-Netz links befindet sich in einem stabilen Zustand. Dies bedeutet die Markierungen der Plätze ändert sich nicht. Dennoch feuern alle Transitionen und Tokens fließen durchs Netzwerk. Rechts in der Abbildung sind die Tokens dargestellt, die während der Simulation über die beiden Kanten (p_1, t_2) und (p_1, t_3) fließen.

Berechnet werden diese Informationen nur, wenn dies im Modell explizit gewünscht ist. Dafür wurde das Settings-Objekt um den zusätzlichen Parameter `showTokenFlow` erweitert. Der Default-Wert ist `false`. Wenn dieser Wert auf `true` geändert wird, werden die zusätzlichen Variablen und Gleichungen generiert. Die zusätzlichen Variablen umfassen jeweils den Zufluss (`tokenFlow.inflowSum`) und Abfluss (`tokenFlow.outflowSum`) jedes Platzes. Der Tokenfluss jeder Kante des Vorbereichs wird durch `tokenFlow.inflow[i]` ausgedrückt und die des Nachbereichs durch `tokenFlow.outflow[j]`.

Diese Informationen können in einer gewöhnlichen Modelica-Oberfläche, wie z.B. OMEdit, betrachtet werden. Allerdings ist auch hierbei das Mapping der einzelnen Variablen auf die entsprechend Kanten manuell auszuführen. Dies erschwert die Interpretation der Ergebnisse erheblich. Die Oberfläche VANESA hingegen ist in der Lage, die Fluss-Informationen auf die entsprechenden Petri-Netz-Objekte zu beziehen. So kann z.B. direkt eine Kante in der grafischen Modellrepräsentation ausgewählt werden und die entsprechenden Flüsse werden dann links im Vorschaufenster angezeigt.

5.3 Erweiterung um gefaltete Plätze

Aufbauend auf den Definitionen aus Abschnitt 2.2 „Petri-Netze“ werden im Folgenden gefaltete Plätze für diskrete und kontinuierliche Petri-Netze eingeführt. Ein gefalteter Platz kann Token unterschiedlicher Sorten enthalten. Üblicherweise werden diese verschiedenen Sorten auch Farben genannt. Somit kann ein gefalteter Platz mehr Informationen als ein herkömmlicher Platz enthalten.

Zur Veranschaulichung soll nachfolgend das Beispiel eines Aquariums dienen, dessen Wasser zum Teil durch Frischwasser ersetzt werden soll. Abbildung 5.4 zeigt ein kontinuierliches Petri-Netz, das den Zu- und Ablauf beschreibt. Hierfür wird jeweils eine Transition verwendet, deren maximale Geschwindigkeitsfunktion den Wasserfluss beschreibt und im Beispiel konstant mit $v = 1$ angenommen wird. Die Wassermenge in dem Aquarium (p_1) und die des herausgepumpten Wassers (p_2) werden jeweils durch einen kontinuierlichen Platz beschrieben.

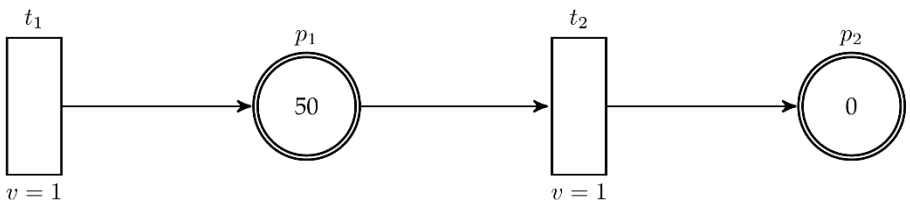

Abbildung 5.4: Beispiel „Aquarium“; der Platz p_1 beschreibt die Wassermenge im Aquarium und der Platz p_2 die abgeführte Wassermenge. Die Transition t_1 beschreibt den Frischwasserzufluss und die Transition t_2 den Wasserabfluss jeweils mit einer maximalen Geschwindigkeit von 1.

Dieses einfache Modell kann genutzt werden, um z.B. die Füllmenge des Aquariums während des Wasseraustauschs zu untersuchen. Für anspruchsvollere Fragestellungen enthält dieses einfache Modell zu wenig Information, wie Folgende zeigen:

- Wann sind 50% des Aquariums mit Frischwasser gefüllt?
- Wieviel Frischwasser wird bei dem Wassertausch mit abgeführt?

Um diese Fragen beantworten zu können, muss das Modell um zusätzliche Informationen und Vorgänge erweitert werden. Die weiteren Informationen sind hier die beiden „Wassersorten“ Frischwasser und Aquariumwasser. Die zusätzlichen Vorgänge sind der differenzierte Zu- und Abfluss für beide „Wassersorten“.

Die Erweiterung des ursprünglichen Modells sieht wie folgt aus:

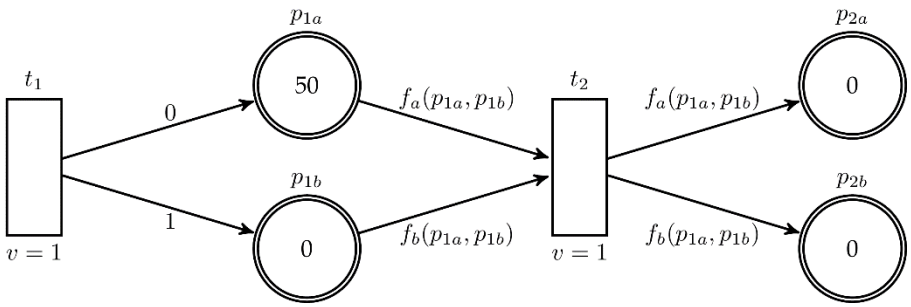

Abbildung 5.5: Ungefaltete Darstellung des Beispiels „Aquarium“.

Diese Art der Modellerweiterung hat zwei entscheidende Nachteile:

1. Die Modellstruktur wird deutlich komplexer. Dies ist in diesem Beispiel noch kein Problem, allerdings für biologische Anwendungen, die häufig eine Vielzahl an Reaktionen und Metaboliten umfassen, geht hierdurch die Anschaulichkeit und somit ein entscheidender Mehrwert des Netzes verloren.
2. In der erweiterten Form gehören immer zwei Plätze und zwei Kanten zusammen, da sie als Paar die Information des ursprünglichen Modells verkörpern. Dieser Zusammenhang wird durch das erweiterte Modell aus Abbildung 5.5 nicht wiedergegeben.

Gefaltete Plätze erlauben es, das Modell ohne strukturelle Änderungen um die zusätzliche benötigten Informationen anzureichern. Zudem werden die Plätze, die zusammengehören, als ein gefalteter Platz dargestellt genauso wie die zusammengehörigen Kanten. Der Vorteil dieser Art der Modellierung besteht darin, dass die Struktur des Ausgangsmodells unverändert bleibt, wie der Vergleich von Abbildung 5.4 und Abbildung 5.6 zeigt.

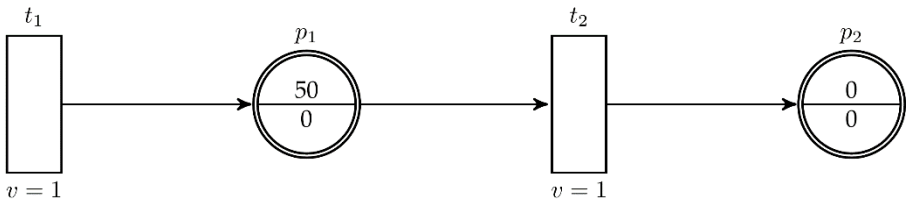

Abbildung 5.6:Beispiel „Aquarium“ aus Abbildung 5.4 mit gefalteten Plätzen modelliert. Jeder Platz speichert nun zwei Informationen. Die obere Zahl steht für das Aquariumwasser und die untere Zahl für das Frischwasser.

Auf die Kantengewichte wird im nächsten Abschnitt näher eingegangen.

Das so erzeugte Netz kann nun genutzt werden, um die oben aufgestellten Fragen zu beantworten. Die Summe der beiden Farben des Platzes p_1 ergeben während der gesamten Simulation konstant 50. Es kann nun zusätzlich der Anteil an Frischwasser separat betrachtet werden:

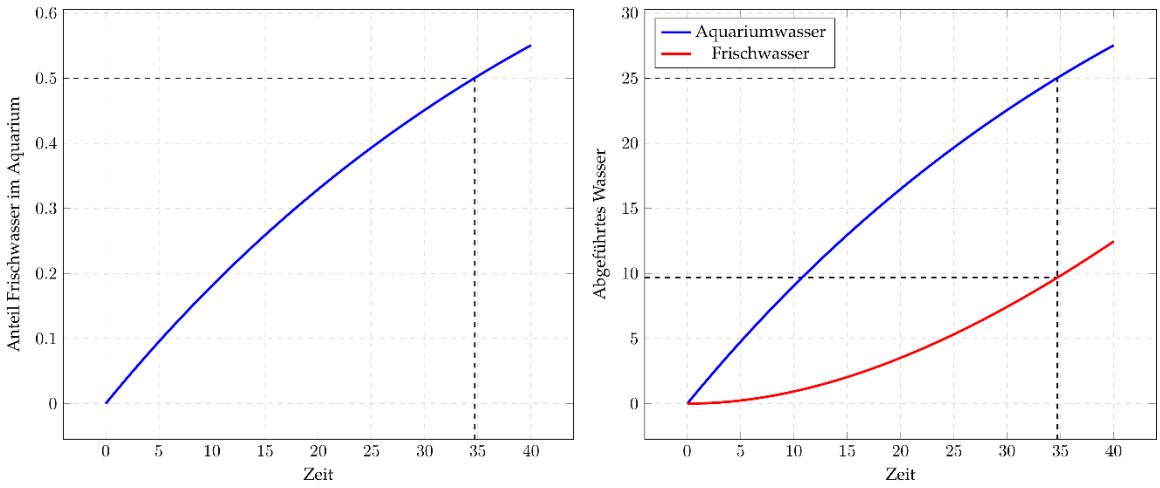

Abbildung 5.7: Simulationsergebnisse zu dem Petri-Netz aus Abbildung 5.6. Links ist der Anteil an Frischwasser im Aquarium zu sehen. Rechts sind die beiden Farben von Platz P2 dargestellt, die beschreiben, wie viel des abgeführten Wassers ursprünglich aus dem Aquarium (blau) bzw. dem Frischwasserzulauf (rot) stammen.

Mit diesen Simulationsergebnissen lassen sich die zuvor aufgestellten Fragen beantworten:

- Nach ca. 35s ist die Hälfte des Wassers durch Frischwasser ersetzt worden.
- In dieser Zeit wurden knapp 10L an Frischwasser abgeführt.

Nachfolgend werden gefaltete Plätze formal definiert und eine Transformation in herkömmlichen Petri-Netzen beschrieben.

5.3.1 Mathematische Definition von Petri-Netzen mit gefalteten Plätzen

Die nachfolgende Definition diskreter Petri-Netze mit gefalteten Plätzen orientiert sich im Wesentlichen an einfach attribuierten gefärbten Petri-Netzen aus [19, p. 64], die wiederum eine Teilmenge der gefärbten Petri-Netze darstellen.

Hierfür werden zunächst die diskreten Petri-Netze aus Definition 2.2 um die Mächtigkeit k (Anzahl der Farben) der Plätze erweitert. Die Mächtigkeit der Plätze gilt für das gesamte Netz. Die Definition 2.2 kann als Spezialfall mit $k = 1$ interpretiert werden. Zusätzlich wird die Gewichtungsfunktion dahingehend abgeändert, dass sie jeder Kante im Netz einen Vektor mit k Elementen als Kantengewicht zuweist:

Definition 5.1: diskretes Petri-Netz mit gefalteten Plätzen (vgl. [19, p. 64])

Ein (diskretes) **Petri-Netz mit gefalteten Plätzen** ist ein Tupel $N = (P, T, F, B, \boldsymbol{f}, k)$, wobei:

a) das 4-Tupel (P, T, F, B) ein Netz gemäß Definition 2.1 ist.
b) $k \in \mathbb{N}$ die **Mächtigkeit** der gefalteten Plätze definiert.
c) die Abbildung $\boldsymbol{f}: F \cup B \to \mathbb{N}_0^k \setminus \{\mathbf{0}\}$ **gefärbte Gewichtungsfunktion** heißt und jedem Pfeil $(p, t) \in F$ bzw. $(t, p) \in B$ einen Zeilenvektor aus nichtnegativen ganzen Zahlen als Gewicht zuweist (allerdings ungleich dem Nullvektor).

Die Definition 2.4 für den Zustand eines Petri-Netzes muss nun um die Mächtigkeit der Plätze erweitert werden:

Definition 5.2: Zustand eines Petri-Netz mit gefalteten Plätzen (vgl. [19, p. 66])

Es sei ein Petri-Netz mit gefalteten Plätzen $N = (P, T, F, B, \boldsymbol{f}, k)$ gegeben. Eine Abbildung $\boldsymbol{z}: P \to \mathbb{N}_0^k$ heißt **Zustand** oder **Markierung** von N, wobei $\boldsymbol{z}$ eindeutig als Vektor $\boldsymbol{z} = (\boldsymbol{z}_1, \dots, \boldsymbol{z}_m)^\top$ mit $\boldsymbol{z}_i = \boldsymbol{z}(p_i)$ darstellbar ist, welcher als Matrix interpretiert werden kann.
Zusätzlich sei $z_{ij} \coloneqq \boldsymbol{z}(p_i, j)$ die Markierung der j-ten Farbe und $|\boldsymbol{z}(p_i)|$ die Summe der Markierung aller Farben des Platzes p_i.
Die Menge Z_r aller Abbildungen von P in $\mathbb{N}_0^k$ wird der **Zustandsraum** von N genannt.

Die Definition 2.5 für das serielle Feuern muss ebenso um die Mächtigkeit der Plätze erweitert werden. Dies bedeutet im Wesentlichen, dass für jede Farbe die Feuerbarkeit geprüft werden muss:

Definition 5.3: Serielles Feuern bei Petri-Netz mit gefalteten Plätzen (vgl. [19, p. 67])

Es sei $N = (P, T, F, B, \boldsymbol{f}, k)$ ein Petri-Netz mit gefalteten Plätzen und $\boldsymbol{z}$ ein Zustand von N.

a) Eine Transition $t \in T$ von N heißt **aktiviert** oder **seriell feuerbar** im Zustand $\boldsymbol{z}$, wenn für alle $p \in {}^\circ t$ gilt: $\boldsymbol{f}(p, t) \leq \boldsymbol{z}(p)$.
b) Eine im Zustand $\boldsymbol{z}$ aktivierte Transition t wird kurz auch $\boldsymbol{z}$**-feuerbar** genannt.
c) Das **Feuern** einer $\boldsymbol{z}$-feuerbaren Transition $t \in T$ von N ist der Übergang vom Zustand $\boldsymbol{z} = (\boldsymbol{z}_1, \dots, \boldsymbol{z}_m)^\top$ in den Zustand $\boldsymbol{z}' = (\boldsymbol{z}'_1, \dots, \boldsymbol{z}'_m)^\top$, wobei für $i = 1, \dots, m$ gilt:

$$\boldsymbol{z}'_i = \begin{cases} \boldsymbol{z}_i - \boldsymbol{f}(p_i, t) + \boldsymbol{f}(t, p_i), & p_i \in {}^\circ t \wedge p_i \in t^\circ \\ \boldsymbol{z}_i - \boldsymbol{f}(p_i, t), & p_i \in {}^\circ t \wedge p_i \notin t^\circ \\ \boldsymbol{z}_i + \boldsymbol{f}(t, p_i), & p_i \notin {}^\circ t \wedge p_i \in t^\circ \\ \boldsymbol{z}_i, & p_i \notin {}^\circ t \wedge p_i \notin t^\circ \end{cases}$$

Analog hierzu muss auch die Definition 2.6 für das nebenläufige Feuern um die Mächtigkeit der gefalteten Plätze erweitert werden. Dies orientiert sich an Definition 4.9 aus [19].

Definition 5.4: Nebenläufiges Feuern bei Petri-Netz mit gefalteten Plätzen (vgl. [19, p. 77])

Es sei $N = (P, T, F, B, \boldsymbol{f}, k)$ ein Petri-Netz mit gefalteten Plätzen im Zustand $\boldsymbol{z}$ und $T_n \subseteq T$ eine Menge von Transitionen. Mit ${}^\circ T_n$ werde der gemeinsame Vorbereich der Transitionen T_n bezeichnet. Die Transitionsmenge T_n heißt **nebenläufig feuerbar** im Zustand $\boldsymbol{z}$, wenn alle Transitionen aus T_n seriell feuerbar sind, und für alle $p \in {}^\circ T_n$ gilt:

$$\sum_{t \in T_n \cap p^\circ} \boldsymbol{f}(p, t) \leq \boldsymbol{z}(p).$$

Der Zustand $\boldsymbol{z}'_i$ eines Platzes $p_i \in P$ wird nach der folgenden Gleichung bestimmt:

$$\boldsymbol{z}'_i = \boldsymbol{z}_i - \sum_{t \in T_n \cap p_i{}^\circ} \boldsymbol{f}(p_i, t) + \sum_{t \in T_n \cap {}^\circ p_i} \boldsymbol{f}(t, p_i).$$

Ebenso lassen sich Petri-Netze mit Konfliktlösung auf Petri-Netze mit gefalteten Plätzen übertragen:

Definition 5.5: (diskretes) Petri-Netz mit gefalteten Plätzen mit Konfliktlösung (vgl. [19, p. 40f])

Das Tupel $(P, T, F, B, \boldsymbol{f}, k, \boldsymbol{z}, \alpha, \delta, \zeta)$ ist ein **Petri-Netz mit gefalteten Plätzen mit Konfliktlösung**, wenn gilt:

a) $(P, T, F, B, \boldsymbol{f}, k)$ ist ein Petri-Netz mit gefalteten Plätzen gemäß Definition 5.1 im Zustand $\boldsymbol{z}$.

b) Die Abbildung $\alpha: P \to \{\text{Priorität}, \text{Wahrscheinlichkeit}\}$ ist eine **Bewertungsartfunktion**, welche jedem Platz $p \in P$ eine bestimmte Art von Bewertung zuordnet.

c) Die Abbildung $\delta: F \to \begin{cases} \mathbb{N}, & \alpha(p) = \text{Priorität} \\ [0,1] \subset \mathbb{R}, & \alpha(p) = \text{Wahrscheinlichkeit} \end{cases}$ ist eine **Bewertungsfunktion**, welche jedem Pfeil von einem Platz $p \in P$ zu einer Transition $t \in p^\circ$ eine Bewertung $\delta(p, t)$ entsprechend der Bewertungsart des Platzes und unter den folgenden Bedingungen zuordnet:

   a. Jede Priorität darf nur einmal von jedem Platz p benutzt werden:
   $$\delta(p, t_i) \neq \delta(p, t_j)\ \forall t_i, t_j \in p^\circ \text{ mit } t_i \neq t_j \text{ falls } \alpha(p) = \text{Priorität}.$$

   b. Die Summe der Wahrscheinlichkeiten von einem Platz p zu den Transitionen $t \in p^\circ$ muss gleich 1 sein:
   $$\sum_{t \in p^\circ} \sigma(p, t) = 1\ \forall p \text{ falls } \alpha(p) = \text{Wahrscheinlichkeit}.$$

Bis hierhin enthalten die Petri-Netze mit gefalteten Plätzen keinen Zeitbegriff. So konnten sie zwar durch das Feuern von einem Zustand in einen Folgezustand wechseln, doch diese Zustände stehen lediglich als Abfolge in Bezug zueinander. Dies ändert sich durch die Einführung eines globalen Zeitbegriffs, analog zu Definition 2.8:

Definition 5.6: Zeitbasiertes Petri-Netz mit gefalteten Plätzen (vgl. [1, p. 100])

Das Tupel $(P, T, F, B, \boldsymbol{f}, k, \boldsymbol{z}, \alpha, \delta, \zeta, d, time)$ ist ein **zeitbasiertes Petri-Netz mit gefalteten Plätzen**, wenn gilt:

a) $(P, T, F, B, \boldsymbol{f}, k, \boldsymbol{z}, \alpha, \delta, \zeta)$ ist ein Petri-Netz mit gefalteten Plätzen mit Konfliktlösung gemäß Definition 5.5.

b) Die Abbildung $d: T \to \mathbb{R}^+$ ist eine **Verzögerungsfunktion**, die jeder Transition $t \in T$ eine nicht-negative Zahl zuordnet, die die Verzögerung der jeweiligen Transition beschreibt.

c) $time \coloneqq [a, b] \subseteq \mathbb{R}$ ist die Menge aller Zeitpunkte, kurz Zeit genannt. a ist der initiale Zeitpunkt und b der finale Zeitpunkt.

In einem weiteren Schritt werden die Petri-Netze mit gefalteten Plätzen um funktionale Kanten erweitert. Damit kann einer Kante fortan nicht nur ein konstantes Kantengewicht zugewiesen werden, sondern auch ein funktionaler Ausdruck:

Definition 5.7: Funktionales Petri-Netz mit gefalteten Plätzen (vgl. [1, p. 97])

Das Tupel $N = (P, T, F, B, \boldsymbol{f}, k, \boldsymbol{z}, \alpha, \delta, \zeta, d, time)$ ist ein funktionales Petri-Netz, wenn gilt:

a) $(P, T, F, B, \boldsymbol{f}, k, \boldsymbol{z}, \alpha, \delta, \zeta, d, time)$ ist ein zeitbasiertes Petri-Netz mit gefalteten Kanten mit Konfliktlösung gemäß Definition 5.6.

b) Die Gewichtungsfunktion $\boldsymbol{f}$ ist dahingehend modifiziert, dass $\boldsymbol{f}: (F \cup B, Z_r, time) \rightarrow \mathbb{N}_0^k$ eine **gefärbte funktionale Gewichtungsfunktion** ist, die zusätzlich von dem Zustandsraum von N und der Zeit abhängt.

Bei den funktionalen Petri-Netzen mit gefalteten Plätzen ist es wichtig, dass die 0 als Kantengewicht zuzulassen ist. Ansonsten könnte innerhalb eines Netzes keine Transition feuern, die im Vorbereich lediglich von einer Farbe „gefüttert“ wird. Siehe hierzu auch das Petri-Netz aus Abbildung 5.8.

Definition 5.8: Kontinuierliches Petri-Netz mit gefalteten Plätzen (vgl. [1, p. 107])

Das Tupel $(P, T, F, B, \boldsymbol{f}, k, v, \boldsymbol{z}, time)$ ist ein **kontinuierliches Petri-Netz mit gefalteten Plätzen**, wenn gilt:

a) Das Tupel (P, T, F, B) ist ein Netz gemäß Definition 2.1.

b) $k \in \mathbb{N}$ die Mächtigkeit der gefalteten Plätze definiert.

c) $\boldsymbol{f}: (F \cup B, Z_r, time) \rightarrow \mathbb{R}_0^{+k}$ ist **gefärbte funktionale Gewichtungsfunktion**.

d) $\boldsymbol{z}$ ist der Zustand des Petri-Netzes.

e) $v: T \rightarrow \mathbb{R}_0^+$ ist die **maximale Geschwindigkeitsfunktion**, die jeder Transition $t \in T$ die maximale Geschwindigkeit $v(t)$ zuordnet.

Mit diesen Definitionen lässt sich das Beispiel „Aquarium“ vollständig formal beschreiben, wie Abbildung 5.8 zeigt:

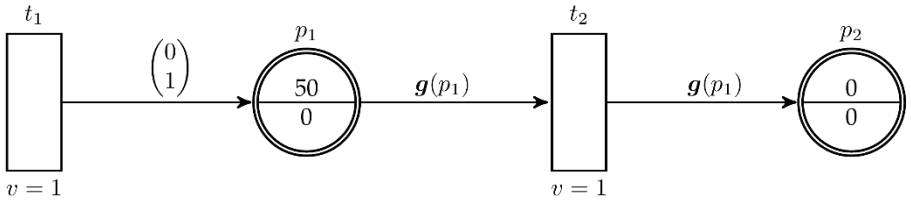

Abbildung 5.8: Beispiel "Aquarium" als Petri-Netz mit gefalteten Plätzen und gefärbten funktionalen Gewichtungsfunktionen.

Die Abbildung $\boldsymbol{g}: P \rightarrow \mathbb{R}_0^{+2}$, die in Gleichung (5.1) definiert ist und an den Kanten des Vor- und Nachbereichs der Transition t_2 steht, beschreibt die perfekte Durchmischung des Wassers im Aquarium. Ihre Einträge summieren sich zu 1. Jede Farbe erhält ihr Gewicht proportional zu der vorhandenen Markierung von p_1.

$$\boldsymbol{g}(p) \coloneqq \begin{cases} 0{,}5 \cdot \mathbf{1}, & \boldsymbol{z}(p)^\top \boldsymbol{z}(p) = 0 \\ \dfrac{1}{|\boldsymbol{z}(p)|} \cdot \boldsymbol{z}(p), & sonst \end{cases} \tag{5.1}$$

5.3.2 Entfaltung von Petri-Netzen mit gefalteten Plätzen

Ein Petri-Netz auf Basis des xHPN-Formalismus mit gefalteten Plätzen kann immer durch Entfaltung in ein äquivalentes ungefärbtes Petri-Netz transformiert werden. Dadurch ist es nicht notwendig, die Konfliktlösungsstrategien neu zu definieren, sondern es kann auf die bestehenden Algorithmen zurückgegriffen werden. Die nachstehende Definition 5.9 beschreibt diese Entfaltung formal:

Definition 5.9: Entfaltung eines Petri-Netzes mit gefalteten Plätzen

Es sei $N = (P, T, F, B, \boldsymbol{f}, k)$ **Petri-Netz mit gefalteten Plätzen** und $N^* = (P^*, T^*, F^*, B^*, f^*)$ das zugehörige entfaltete ungefärbte Petri-Netz.

a) Für jeden Platz $p \in P$ werden im entfalteten Petri-Netz k Plätze erstellt $\{p^1, p^2, \dots, p^k\}$: $P^* := \bigcup_{i=1}^{m} \{p_i^1, p_i^2, \dots, p_i^k\}$.

b) Die Transitionen bleiben unverändert: $T^* := T$.

c) Für jede Kante (p_i, t) bzw. (t, p_i) des Vor- bzw. Nachbereichs werden k Kanten (p_i^j, t) bzw. (t, p_i^j) mit $j = 1,2, \dots, k$ erstellt.

d) $f^*(p_i^j, t)$ bzw. $f^*(t, p_i^j)$ wird die j-te Komponente der Gewichtungsfunktion $\boldsymbol{f}(p_i, t)$ bzw. $\boldsymbol{f}(t, p_i)$ zugewiesen.

5.3.3 Implementierung

Die Implementierung der gefärbten Plätze erfolgt über Matrix-Konnektoren (siehe Listing 5.1, Zeilen 15-16). Dies erlaubt, die `connect`-Gleichungen eines ungefärbten Netzes unverändert zu übernehmen und lediglich die Komponentendefinition auf die Färbung anzupassen.

```
1  within PNlib.Examples.Models.ColoredPlaces;
2  model CPC
3    Real color[numColors] "marking";
4    Real t = sum(color) "total marking";
5    parameter Integer nIn=0 "number of input transitions";
6    parameter Integer nOut=0 "number of output transitions";
7
8    parameter Real startMarks[numColors] = fill(0, numColors)
9      "start marks";
10   parameter Real minMarks[numColors] = fill(0, numColors)
11     "minimum capacity";
12   parameter Real maxMarks[numColors] = fill(PNlib.Constants.inf,
13     numColors) "maximum capacity";
14
15   PNlib.Interfaces.PlaceIn[nIn, numColors] inTransition;
16   PNlib.Interfaces.PlaceOut[nOut, numColors] outTransition;
17   Integer animateMarking=settings.animateMarking
18     "only for place animation and display (Do not change!)";
19 protected
20   outer PNlib.Settings settings
21     "global settings for animation and display";
```

```
22    PC places[numColors](nIn=fill(nIn, numColors), nOut=fill(
23      nOut, numColors), startMarks=startMarks, minMarks=minMarks,
24      maxMarks=maxMarks);
25  equation
26    color = places.t;
27
28    for i in 1:nIn loop
29      for j in 1:numColors loop
30        connect(inTransition[i,j], places[j].inTransition[i]);
31      end for;
32    end for;
33
34    for i in 1:nOut loop
35      for j in 1:numColors loop
36        connect(outTransition[i,j], places[j].outTransition[i]);
37      end for;
38    end for;
39  end CPC;
```

Listing 5.1: PNlib Komponente CPC für einen gefärbten kontinuierlichen Platz.

Da die Konnektoren von Plätzen und Transitionen kompatibel sein müssen, mussten alle Petri-Netz-Komponenten (`PNlib.PC`, `PNlib.PD`, `PNlib.TC`, `PNlib.TD`, etc.) in einer gefärbten Variante implementiert werden. Letztendlich sind dies Wrapper, die die entsprechenden Elemente der Matrix-Konnektoren auf die ungefärbten Komponenten abbilden (vgl. Listing 5.1, Zeilen 28-38). Listing 5.1 zeigt exemplarisch die Umsetzung der Komponente `CPC` für gefärbte kontinuierliche Plätze in der PNlib.

5.4 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie die Modelica-Bibliothek PNlib ausgehend von der Version 1.0 weiterentwickelt wurde. In Version 1.0 wurde die Bibliothek lediglich von einem kommerziellen Modelica-Tool unterstützt. Inzwischen haben sich sowohl die Sprache Modelica als auch die Modelica-Programme weiterentwickelt, sodass die PNlib 1.0 inzwischen von keinem aktuellen Programm mehr unterstützt wird. Die ersten Modifikationen an der PNlib zielten daher darauf ab, die Bibliothek dem Sprachstandard anzupassen und toolspezifische Elemente zu entfernen. Dadurch kann die Bibliothek mit verschiedenen Modelica-Umgebungen und der aktuellen Modelica-Version verwendet werden.

Um sowohl die Entwicklung der Bibliothek als auch die Entwicklung der Modelica-Umgebungen verfolgen und einschätzen zu können, wurde eine Testumgebung für Dymola und OpenModelica entwickelt. Mit dieser können alle Beispielmodelle automatisch ausgeführt und anhand von Musterlösungen auf Richtigkeit geprüft werden.

In einem weiteren Schritt wurde die Bibliothek um weitere Funktionalitäten ergänzt. So ist es nun möglich, Flüsse in Petri-Netzen direkt zu analysieren. Hierfür wird für jeden Platz und jede Kante der entsprechende Fluss während der Simulation summiert.

Zudem wurde die Bibliothek um gefaltete Plätze erweitert. Diese basieren auf den ursprünglichen elementaren Petri-Netz-Elementen und erlauben es, ein Netz um zusätzliche Informationen anzureichern. Die nachfolgende biologische Anwendung eines metabolischen Systems zeigt die Einsatzmöglichkeit dieser zusätzlichen Bibliotheks-Funktionalitäten.

6 Anwendung: Glykolyse in Saccharomyces cerevisiae

In diesem Kapitel wird die praktische Umsetzung eines metabolischen Prozesses anhand eines biologischen Netzes demonstriert, das einen Teil der Glykolyse von Hefezellen beschreibt. Das hier verwendete Modell ist Teil eines größeren Netzwerks, das in Kooperation mit den beiden Arbeitsgruppen „Algae Biotechnology & Bioenergy“ und „Proteom- und Metabolomforschung“ der Universität Bielefeld im Rahmen des MoRitS-Projektes entstanden ist. Die nachfolgenden Ausführungen demonstrieren, wie die zuvor erarbeiteten Methoden und Werkzeuge die Arbeit mit biologischen Netzen unterstützen und dafür genutzt werden können, um mehr Informationen aus den Simulationen zu gewinnen.

Das Teilmodell der Glykolyse, auf dem dieses Kapitel beruht, basiert auf dem biologischen Netz von Hynne [23]. Das Ausgangsmodell von Hynne wurde um die nachstehend aufgelisteten Modifikationen abgeändert:

- Die Produkt-Diffusion wurde nicht modelliert. Konkret wurden die Reaktionen 13, 14, 16, 17, 18, 19, 20, 21 und 22 nicht aus dem Modell von Hynne übernommen.
- Eine zusätzliche Konsum-Reaktion wurde für $\mathrm{NAD^+}$ eingeführt, die dies in NADH umwandelt.
- Die Reaktion 1 aus dem Modell von Hynne wurde irreversibel, also lediglich als Zufütterung, umgesetzt.

Das gesamte Modell ist als biologisches Netz in Abbildung 6.1 dargestellt und mit allen Reaktionen, Parametern und Gleichungen ausführlich in Anhang B beschrieben.

Als typisches Szenario für die Untersuchung eines solchen biologischen Netzes wird nachfolgend eine Input/Output-Analyse durchgeführt. Dies bedeutet, es wird der Einfluss des Modell-Inputs (hier z.B. $\mathrm{Glc_x}$) auf die Endproduktproduktion (hier Glyc und EtOH) untersucht. Vereinfacht lässt sich das Netzwerk durch einen *Inputstrang* und zwei *Outputstränge* beschreiben, die über mehrere Zwischenmetaboliten miteinander verbunden sind. Diese Zwischenmetaboliten werden im Weiteren als *Kernsystem* bezeichnet.

Das Kernsystem besteht aus den Metaboliten FBP, GAP, DHAP, BPG und PEP. Diese sind über reversible Reaktionen miteinander verbunden. So kann im Grunde aus jedem dieser Metaboliten jedes der übrigen gebildet werden. Aus diesem Kernsystem erfolgt die Stoffproduktion von Glyc und EtOH durch die zwei Ausgänge PK und lpGlyc.

Das Kernsystem enthält einen Eingang, die Reaktion PFK. Diese Reaktion wird zum einen durch hohe Startstoffmengen innerhalb des Inputstranges, den Metaboliten $\mathrm{Glc_x}$, Glc, $\mathrm{G6P}$ und $\mathrm{F6P}$, angetrieben und zum anderen durch die Zufütterung von $\mathrm{Glc_x}$ durch die Reaktion inGlc.

Nachfolgend wird untersucht, wie sich der Endproduktaufbau (Ethanol und Glycerin) in direkter Abhängigkeit von der Zufütterung des Kernsystems darstellt.

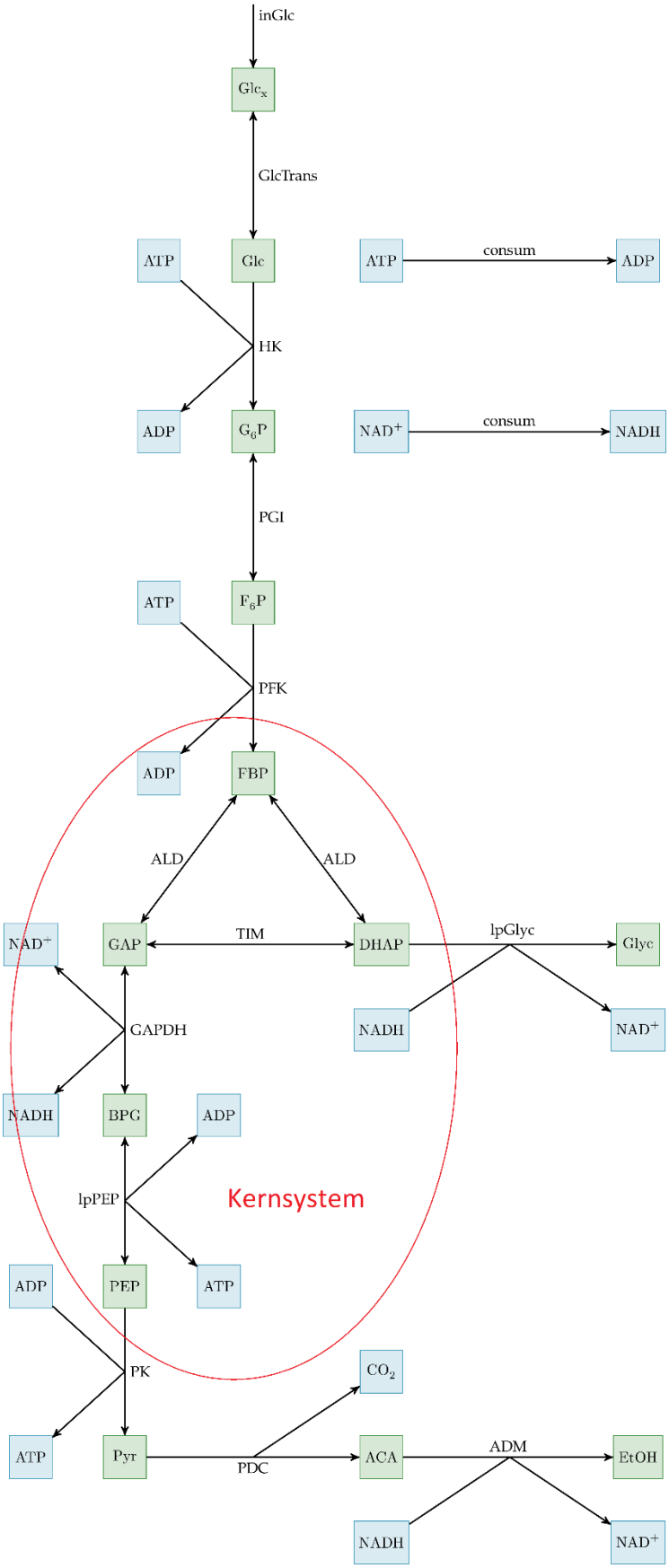

Abbildung 6.1: Das biologische Netz zu dem Teilmodell der Glykolyse von Hefezellen. Die Metaboliten sind grün dargestellt. Der Inputstrang besteht aus Glc_x, Glc, G6P *und* F6P *und das Kernsystem besteht aus* FBP, GAP, DHAP, BPG *und* PEP. *Die Endprodukte sind* Glyc *und* EtOH.

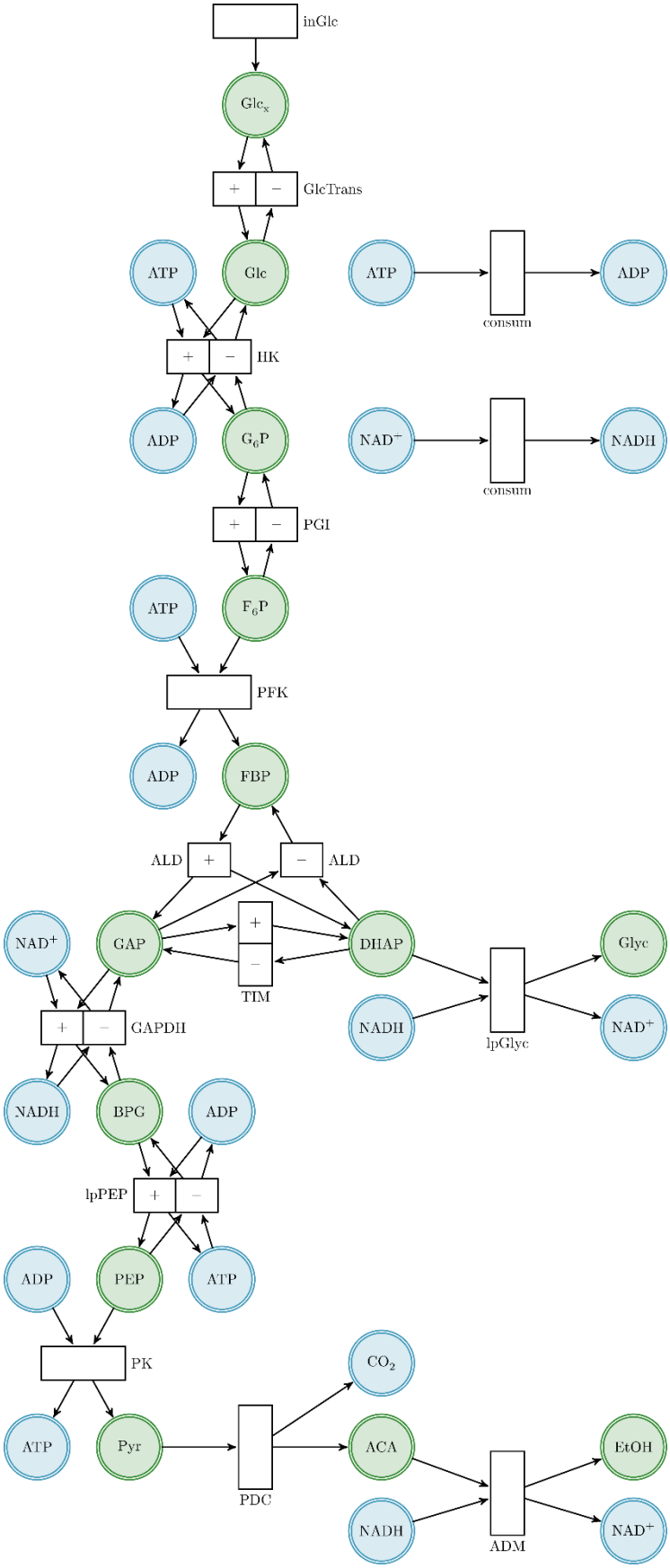

Abbildung 6.2: Das kontinuierliche Petri-Netz zu dem Teilmodell der Glykolyse von Hefezellen aus Abbildung 6.1. Bei den reversiblen Reaktionen, sind die Transitionen der Hinreaktion mit einem Plus und die Reaktionen der Rückreaktion mit einem Minus gekennzeichnet.

6.1 Ungefaltete Simulation

Für die einfache Simulation des biologischen Netzes (Abbildung 6.1), wird dieses zunächst in das Petri-Netz (Abbildung 6.2) transformiert. Diese Transformation kann manuell oder auch mit Unterstützung durch das Programm VANESA (siehe Kapitel 3.3) erfolgen. Hierbei werden die Knoten des biologischen Netzes als Plätze und die Reaktionen als Transitionen dargestellt. Wie bereits im Kapitel „Grundlagen“ erläutert, werden hierbei irreversible Reaktionen durch eine Transition und reversible Reaktionen durch zwei Transitionen abgebildet.

Das so erstellte Petri-Netz wurde als Modelica-Modell realisiert und zusammen mit der PNlib und dem OpenModelica-Compiler simuliert. Die nachstehenden Kurven aus Abbildung 6.3 und Abbildung 6.4 zeigen den Simulationsverlauf der Ein- und Ausgänge.

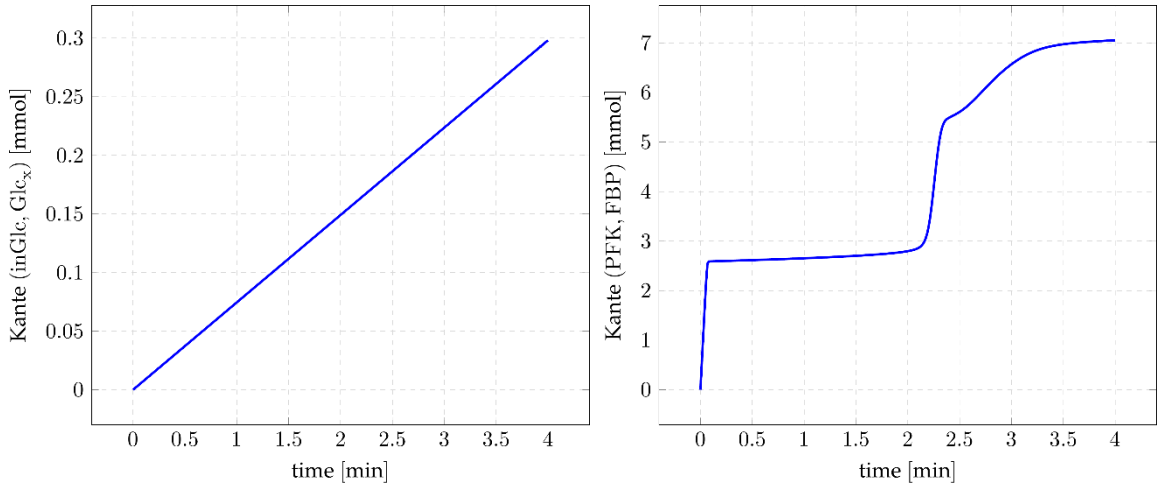

Abbildung 6.3: Die Summe des Tokenflusses über die Kanten (inGlc, Glc_x) *und* (PFK, FBP).

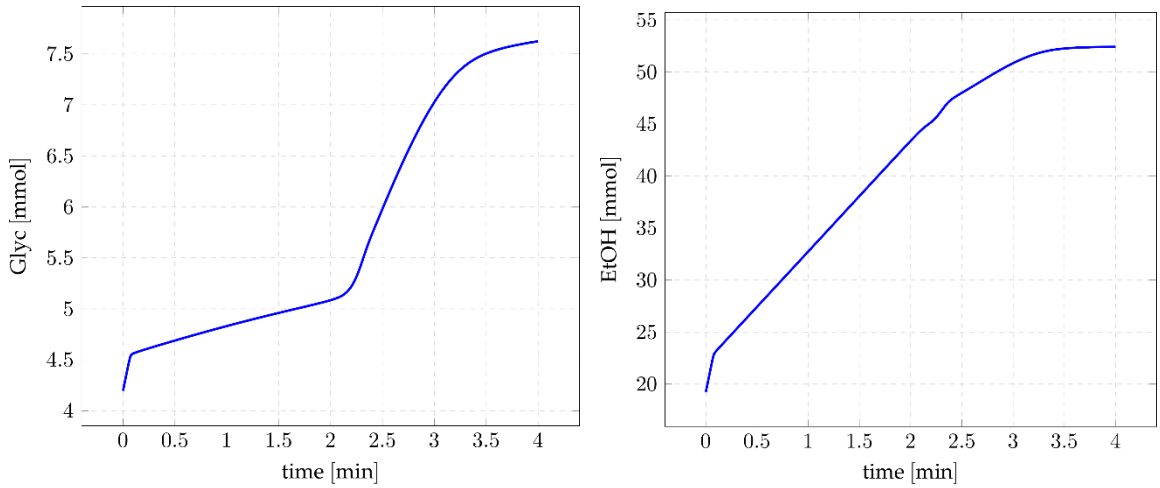

Abbildung 6.4: Stoffmengenverläufe der Endprodukte Glyc *(links) und* EtOH *(rechts).*

Das System wird mit der Reaktion inGlc kontinuierliche gefüttert. Abbildung 6.3 (links) zeigt, dass sich dies für den simulierten Zeitraum von 4 Minuten zu einer Eingabe von $0{,}3mmol$ summiert. Die Stoffeingabe über die Kante (PFK, FBP) in das Kernsystem summiert sich in dem gleichen Zeitraum auf $7{,}1mmol$ (siehe Abbildung 6.3 rechts). Diese Flüsse können dank der in Kapitel 5.2 „Tokenflüsse“ beschriebenen Erweiterung der PNlib nachvollzogen werden.

Aufgrund der Reaktion ALD, siehe Gleichung (6.1), verdoppelt sich die Stoffmenge, die in das Kernsystem eingegeben wird und in die Produkte umgewandelt werden kann. Somit ist mit einem maximalen Produktanstieg von $14{,}8mmol$ auf Basis der Stoffeingabe ins Kernsystem zu rechnen.

$$\mathrm{FBP} \underset{v_6^-}{\overset{v_6^+}{\rightleftharpoons}} \mathrm{DHAP} + \mathrm{GAP} \tag{6.1}$$

Die Stoffmenge Glycerin liegt zu Beginn bei $4{,}2mmol$ und steigt bis zum Ende der Simulation auf $7{,}6mmol$ (siehe Abbildung 6.4 links). Das entspricht einer Zunahme von $3{,}4mmol$. Die Stoffmenge Ethanol steigt während dieses Zeitraums von $19{,}2mmol$ auf $52{,}4mmol$ (siehe Abbildung 6.4 rechts). Das entspricht einer Zunahme von $33{,}2mmol$. Insgesamt ergibt sich somit ein Produktaufbau von $36{,}6mmol$.

Der tatsächliche Produktanstieg um $36{,}6mmol$ setzt sich hauptsächlich aus den Startstoffmengen der Metaboliten des Kernsystems und der Eingabe in das Kernsystem zusammen. Da in dem Kernsystem bereits von Beginn an eine hohe Stoffmenge vorliegt, lässt sich der Produktanstieg nicht direkt auf die Systemeingabe durch den Inputstrang beziehen.

Auch wenn der Produktaufbau nicht direkt auf die Systemeingabe bezogen werden kann, ist es dennoch möglich den Produktaufbau von Glycerin und Ethanol in relativen Anteilen anzugeben, wie Abbildung 6.5 zeigt.

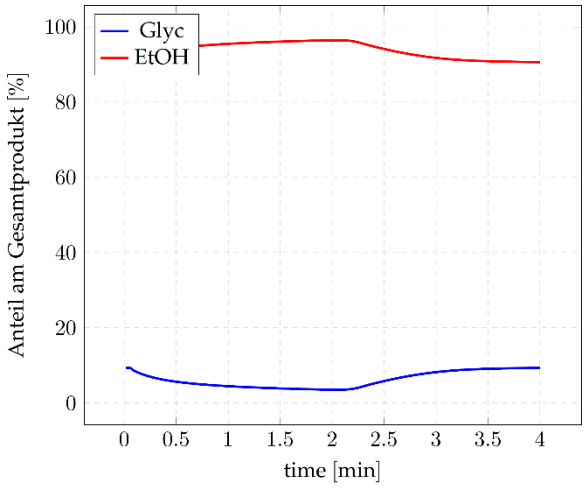

Abbildung 6.5: Produktzuwachs in Anteilen an Glycerin und Ethanol auf Basis des ungefärbten Modells.

Es ist ersichtlich, dass der unmittelbare Einfluss der Eingangsgrößen mit diesem Modell nicht nachvollzogen werden kann. Diese Einschränkung wird im nächsten Abschnitt durch die Faltung der Plätze aufgehoben.

6.2 Gefaltete Simulation

Für eine detailliertere Analyse des biologischen Prozesses können einzelne Metaboliten markiert werden. Im Labor wird hierfür beispielsweise das Kohlenstoffisotop ^{13}C verwendet

[42], das sich im metabolischen Prozess beobachten lässt. Dies macht es möglich, Pathways zu finden, die von dem ^{13}C-markierten Stoff primär genutzt werden. In Anlehnung an dieses Vorgehen können in der Petri-Netz-basierten Simulation gefaltete Plätze verwendet werden. Dies erlaubt eine Zweifärbung der Metaboliten, die genutzt werden kann, um markierte und umarkierte Anteile separat verfolgen zu können. Die Markierung erfolgt, wie im Labor, auf Basis der Kohlenstoffatome, da diese der gemeinsame Bestandteil aller Metaboliten sind. Tabelle 6.1 beschreibt die Metaboliten aus dem obigen Anwendungsfall mit dem entsprechenden Kohlenstoffbestandteil.

Tabelle 6.1: Die Metaboliten aus Abbildung 6.1 mit ihrer jeweiligen Anzahl Kohlenstoffatome.

Metabolit	Name	
Glc	Glucose	C_6-Körper
G6P	Glucose-6-phosphat	C_6-Körper
F6P	Fructose-6-phosphat	C_6-Körper
FBP	Fructose-1,6-bisphosphat	C_6-Körper
GAP	Glycerinaldehyd-3-phosphat	C_3-Körper
DHAP	Dihydroxyacetonphosphat	C_3-Körper
Glyc	Glycerin	C_3-Körper
BPG	2,3-Bisphosphoglycerat	C_3-Körper
PEP	Phosphoenolpyruvat	C_3-Körper
Pyr	Pyruvat	C_3-Körper
ACA	Acetaldehyd	C_2-Körper
EtOH	Ethanol	C_2-Körper

Die Modellbildung des gefalteten Netzes ist auf Basis des ungefärbten Petri-Netzes einfach, da die ursprüngliche Netzstruktur unverändert übernommen werden kann. Lediglich die Plätze und Kanten werden um zusätzliche Information zu der Faltung erweitert. Die einfachen Plätze des ungefärbten Netzes werden durch zweigefärbte Plätze ersetzt. Die Startmarkierung wird für jeden Metaboliten entsprechend der ersten oder zweiten Farbe zugeordnet. Im Folgenden wird der Zusammenhang der Glycerin- und Ethanol-Produktion in Abhängigkeit zu dem Modelleingang, der Kante $(\mathrm{PFK}, \mathrm{FBP})$, untersucht. Entsprechend werden im Ausgangszustand die Metaboliten $\mathrm{Glc_x}$, Glc, $\mathrm{G6P}$ und $\mathrm{F6P}$ markiert, da diese gemeinsam den Modelleingang darstellen.

Neben den Plätzen müssen auch die Kanten um zusätzliche Informationen erweitert werden. Jedes Kantengewicht wird durch die Faltung zu einem zweidimensionalen Vektor, der das ursprünglich skalare Kantengewicht auf die beiden Farben des Platzes aufteilt. Somit ist klar, dass die Summe der Vektorelemente das ursprünglich skalare Kantengewicht ergeben muss. Sofern von einer vollständigen Durchmischung der gefärbten und ungefärbten Kohlenstoffatome ausgegangen werden kann, kann die Aufteilung direkt proportional zu den

Markierungen der entsprechenden Farben geschehen. Diese Durchmischung wird durch die bereits im vorherigen Kapitel eingeführte Abbildung $\boldsymbol{g}$ aus Gleichung (5.1) beschrieben:

$$\boldsymbol{g}(p) \coloneqq \begin{cases} 0{,}5 \cdot \mathbf{1}, & \boldsymbol{z}(p)^{\mathsf{T}}\boldsymbol{z}(p) = 0 \\ \dfrac{1}{|\boldsymbol{z}(p)|} \cdot \boldsymbol{z}(p), & sonst \end{cases}$$

Die Faltung wird nachfolgend exemplarisch anhand der Hinreaktion von dem Enzym ALD gezeigt:

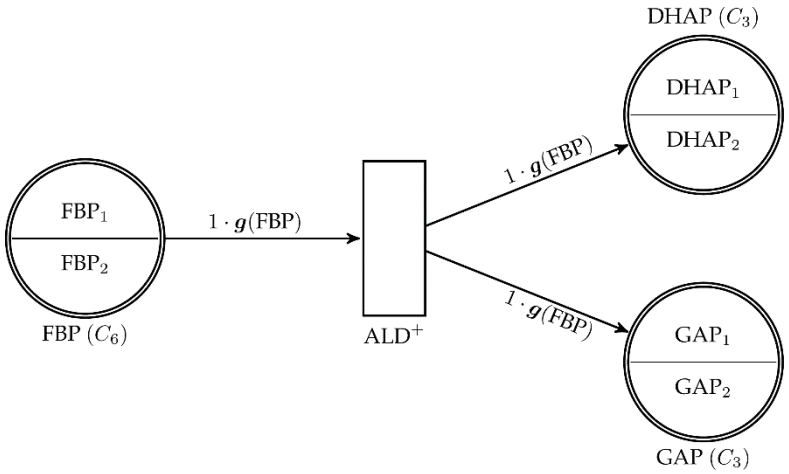

Abbildung 6.6: Hinreaktion ALD^+ mit gefalteten Plätzen. Die Modellierung der Durchmischung beider Farben erfolgt durch die gefärbte funktionale Gewichtungsfunktion an den Kanten.

Die gefärbte funktionale Gewichtungsfunktion an den Kanten setzt sich aus der Abbildung $\boldsymbol{g}$, für die Durchmischung, gewichtet mit der Stöchiometrie der jeweiligen Enzymreaktion zusammen. Die Plätze enthalten jeweils eine markierte Stoffmenge (hier FBP_1, DHAP_1 und GAP_1) sowie eine unmarkierte Stoffmenge (hier FBP_2, DHAP_2 und GAP_2).

Die Simulationsergebnisse aus Abbildung 6.7 des gefalteten Petri-Netzes enthält nun für jeden Metaboliten eine markierte und eine unmarkierte Stoffmenge. Zusammengenommen ergeben beide Stoffmengen die Ergebnisse der ungefärbten Simulation.

Die Stoffmenge Glycerin liegt zu Beginn bei $4{,}2mmol$ und steigt bis zum Ende der Simulation auf $7{,}6mmol$. Das entspricht einer Zunahme von $3{,}4mmol$. Diese Zunahme setzt sich aus $2{,}5mmol$ markiertem und aus $0{,}9mmol$ unmarkiertem Glycerin zusammen. Der Anstieg, der nach der zweiten Minute beginnt, basiert nahezu vollständig auf markiertem Glycerin. Die Kurve des ungefärbten Glycerins ist somit ab diesem Zeitpunkt annähernd konstant.

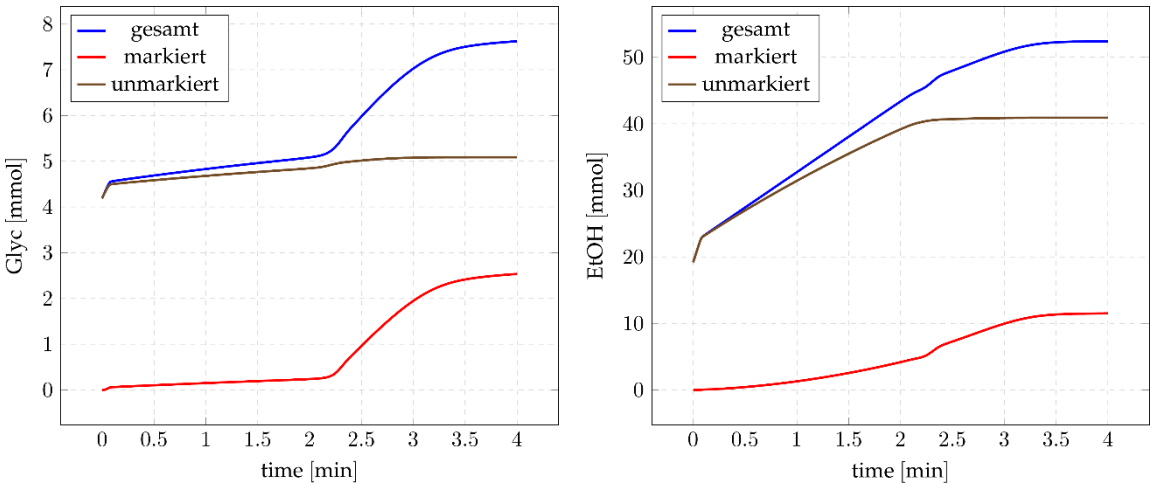

Abbildung 6.7: Stoffmengenverläufe der beiden Produkte Glycerin und Ethanol mit markiertem Substrat Glc_x, Glc, G6P *und* F6P.

Die Stoffmenge Ethanol steigt während der Simulation von $19{,}2mmol$ auf $52{,}4mmol$. Das ist eine Zunahme von $33{,}2mmol$, wobei der markierte Anteil $11{,}5mmol$ beträgt.

Insgesamt ergibt sich ein Produktaufbau von $36{,}6mmol$, wobei der markierte Anteil $14{,}0mmol$ beträgt. Die verbliebenen $0{,}8mmol$ zugeführter Stoffmenge verteilt sich auf die Zwischenmetaboliten und werden nicht in eines der Endprodukte umgewandelt.

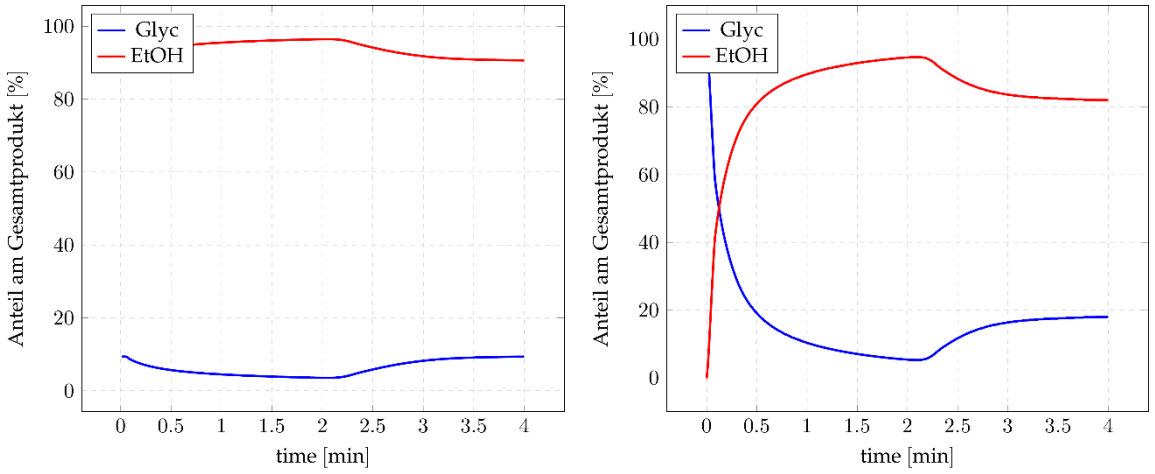

Abbildung 6.8: Produktzuwachs in Anteilen an Glycerin und Ethanol. Links sind die Ergebnisse auf Basis der ungefalteten Simulation und rechts der gefalteten Simulation zu sehen.

Nun kann der Endproduktzuwachs in Bezug zu der Systemeingabe gesetzt werden. Die Ergebnisse unterscheiden sich deutlich zu der Auswertung, die mit der ungefalteten Simulation möglich ist. Abbildung 6.8 stellt die Ergebnisse beider Simulationsansätze gegenüber.

Die qualitativen Kurvenverläufe sind ähnlich. So fällt in beiden Auswertungen der Anteil Glycerin zu Beginn ab. Kurz nach Minute 2 steigt der Glycerin-Anteil leicht an. Die quantitative Aufteilung beider Produktzuwächse unterscheidet sich deutlich. So wird bei der ungefalteten

Simulation ein Produktzuwachs von 9% ermittelt. Durch die Markierung der Eingangsgrößen, kann in der gefalteten Simulation ein aussagekräftiger Wert von 18% bestimmt werden. Die Ergebnisse für Ethanol verhalten sich entsprechend dazu.

Markierung von Glc_x

Durch die Modifikation des markierten Modelleingangs lassen sich diese Ergebnisse verifizieren. Hierfür wird nun nicht der gesamte Eingangsstrang markiert, sondern lediglich Glc_x. Die Startmarkierung von Glc_x liegt bei $1{,}6mmol$ und die Zufütterung während der Simulation liegt bei $0{,}3mmol$ (siehe Abbildung 6.3 links). Dies bedeutet, dass in diesem Durchgang insgesamt $1{,}9mmol$ markierter Metaboliten im Inputstrang und somit aufgrund der Reaktion ALD maximal die doppelte Menge, $3{,}8mmol$, im übrigen System befindet.

Nachfolgend werden die Simulationsergebnisse entsprechend der ersten Simulation aufbereitet und dieser gegenübergestellt:

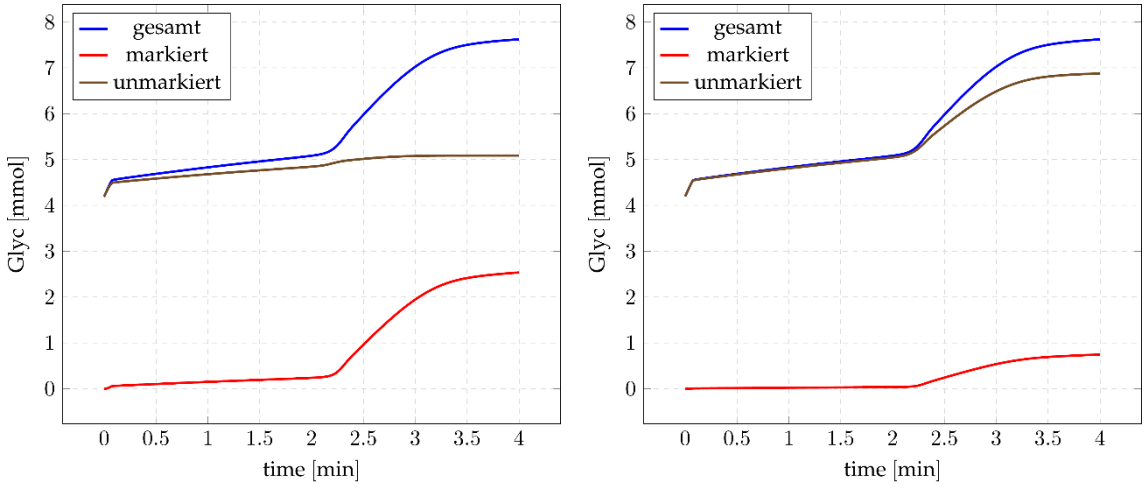

Abbildung 6.9: Gegenüberstellung des Glycerin-Aufbaus; links ist das Ergebnis mit markiertem Inputstrang (siehe auch Abbildung 6.7) und rechts mit markiertem Glc_x dargestellt.

Die Abbildung 6.9 veranschaulicht den Stoffmengenaufbau beider gefärbter Simulationsdurchläufe für das Produkt Glycerin. Die blaue Kurve zeigt die Gesamtstoffmenge, bestehend aus dem markierten und unmarkierten Anteil. Dieser Verlauf ist in beiden Durchläufen identisch und stimmt mit dem Ergebnis der ungefärbten Simulation aus Abbildung 6.4 (links) überein. Die Unterschiede sind lediglich bei der Aufteilung in markierte und unmarkierte Anteile zu sehen. Erwartungsgemäß ist der markierte Anteil des zweiten Durchlaufs geringer als der des ersten Durchlaufs, da auch die Menge des markierten Stoffes im gesamten Modell geringer ist. Bis zum Ende der Simulation bei Minute 4 wurden $0{,}75mmol$ markiertes Glycerin aufgebaut.

Des Weiteren ist zu erkennen, dass sich die Zusammensetzung der Glycerin-Produktion etwa ab dem Zeitpunkt 2,5 Minuten verändert. Im ersten Durchlauf erfolgte der Produktaufbau ausschließlich auf Basis des markierten Stoffes. Im zweiten Durchlauf hingegen, setzt sich der

Produktaufbau bis zum Schluss der Simulation sowohl aus markiertem wie auch unmarkiertem Stoff zusammen.

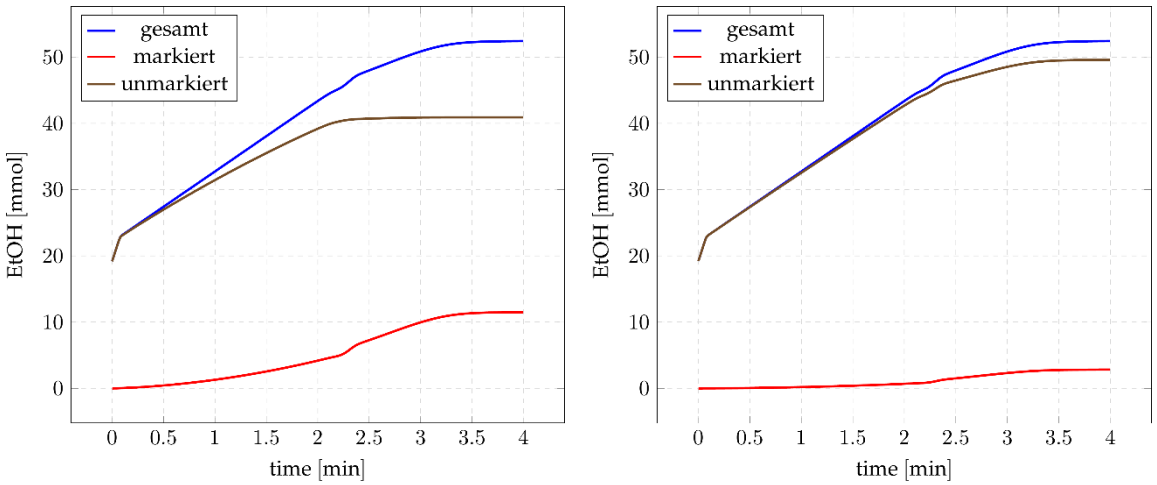

Abbildung 6.10: Gegenüberstellung des Ethanol-Aufbaus; links ist das Ergebnis mit markiertem Inputstrang (siehe auch Abbildung 6.7) und rechts mit markiertem Glc_x dargestellt.

Abbildung 6.10 zeigt, dass neben $0{,}75 mmol$ markiertem Glycerin auch $2{,}84 mmol$ markiertes Ethanol produziert wurde. Zusammen ergibt sich ein Endproduktaufbau von $3{,}59 mmol$. Die übrigen markierten Kohlenstoffatome verteilen sich auf die Zwischenmetaboliten.

Analog zu den Ergebnissen der Glycerin-Produktion zeigen auch die Ethanol-Ergebnisse kein Plateau im unmarkierten Anteil, wenn lediglich $\mathrm{Glc_x}$ und nicht der ganze Inputstrang markiert wird.

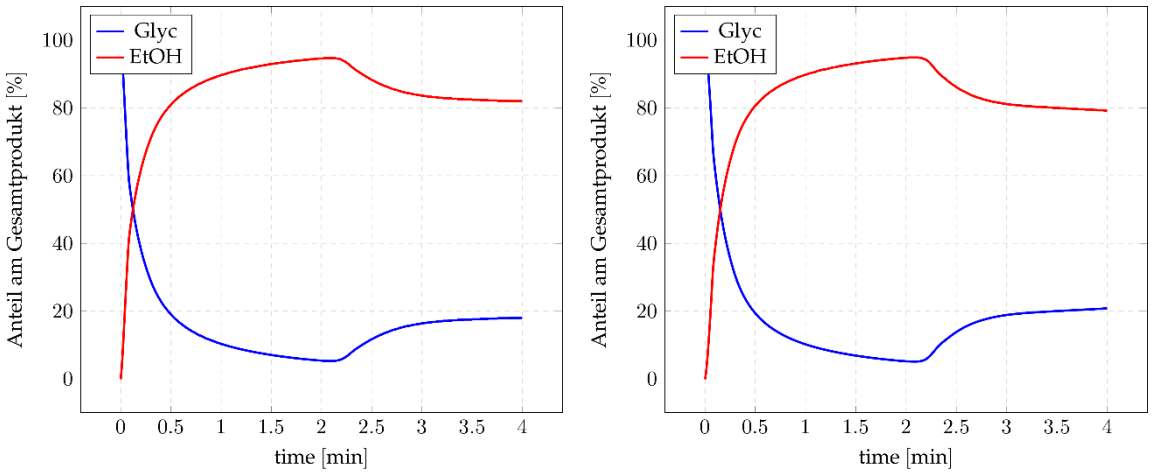

Abbildung 6.11: Gegenüberstellung des Produktzuwachs in Anteilen an Glycerin und Ethanol auf Basis der beiden markierten Modelle. Links ist das Ergebnis mit markiertem Inputstrang (siehe auch Abbildung 6.8 rechts) und rechts mit markiertem Glc_x dargestellt.

Die beiden gefalteten Simulationen führen zu einer sehr ähnlichen Verteilung der Glycerin- und Ethanol-Produktion, wie Abbildung 6.11 zeigt. Der Glycerin-Anteil liegt mit eingefärbtem $\mathrm{Glc_X}$ bei 21% und damit 3 Prozentpunkte über den Ergebnissen mit komplett markiertem Inputstrang.

6.3 Zusammenfassung

Zunächst wurde ein metabolischer Prozess auf Basis eines biologischen Netzes von Hynne [23] in ein kontinuierliches, zeitbasiertes Petri-Netz überführt und simuliert. Auf Basis der Simulationsergebnisse wurde der Zusammenhang zwischen Modelleingang und Modellausgang untersucht. Anschließend wurde das Petri-Netz um gefaltete Plätze ergänzt, sodass die Kohlenstoffatome einzelner Metaboliten markiert und im metabolischen Prozess verfolgt werden konnten. Mit diesem Modell wurden zwei unterschiedliche Startmarkierungen simuliert und die Ergebnisse gegenübergestellt.

Das gefaltete Netz erlaubt es, die Einflüsse bestimmter Metaboliten und Modellteile auf die Simulation zu untersuchen. Dadurch können Zusammenhänge erkannt und nachgewiesen werden.

Mit den durchgeführten Simulationen konnte gezeigt werden, dass es einen deutlichen Unterschied macht, ob der Einfluss des Modellinputs direkt (gefaltete Simulation) oder indirekt (ungefaltete Simulation) untersucht wird.

Darüber hinaus bietet der hier vorgestellte Ansatz zur Kohlenstoffflussanalyse weitere Analysemöglichkeiten für Systeme mit mehreren Inputs. Bei diesen Systemen gibt es die Möglichkeit, den Einfluss einzelner Inputs auf das Gesamtsystem zu untersucht. Ohne die Möglichkeit der Markierung, können solche Systeme lediglich durch das Ausschalten der übrigen Inputs analysiert werden. Dies liefert im Allgemeinen keine verlässlichen Ergebnisse, da über Zwischenprodukte wie z.B. NAD^+, ATP und CO_2 verschiedene Modellteile gekoppelt sein können und somit Inputstränge indirekt Einfluss auf das Gesamtsystem haben können. Somit lassen sich ohne die Kohlenstoffflussanalyse die gewonnenen Erkenntnisse nicht direkt auf das Ausgangssystem übertragen. In diesen Fällen ermöglicht die Simulation mit gefalteten Petri-Netzen aussagekräftige Ergebnisse. So lassen sich die Zusammenhänge nachvollziehen und biologische Prozesse im Ganzen verstehen.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die Modellierungssprache Modelica eignet sich aufgrund ihrer Flexibilität und Offenheit für eine Vielzahl an physikalischen Problemstellungen und hat sich seit 2000 entsprechend schnell und weltweit in akademischen und industriellen Anwendung durchgesetzt. Modelica sowie diverse Bibliotheken (insbesondere die Modelica Standardbibliothek) sind kostenfrei und daher für jedermann nutzbar. In Kombination mit der quelloffenen Umgebung OpenModelica können so Modelica-Modelle unentgeltlich und transparent erstellt, bearbeitet und simuliert werden.

Biologische Prozesse können mit Petri-Netzen abgebildet werden. Dies hat den Vorteil, dass die Prozesse hierdurch mit einem mathematischen Formalismus beschrieben sind. Ein deutlicher Nachteil vieler biologischer Simulationsumgebungen (z.B. Cell Illustrator) ist, dass der zugrundeliegende Petri-Netz-Formalismus nicht transparent ist. Somit ist nicht nachvollziehbar, wie Konfliktsituationen von dem Simulator aufgelöst werden. Dadurch sind die Ergebnisse oft nicht nachvollziehbar.

In dem Kapitel Verwandte Arbeiten wurden Softwarelösungen vorgestellt, mit denen metabolische Prozesse auf Basis von Enzymreaktionen abgebildet und simuliert werden. Diese unterteilen sich in zwei Gruppen: Simulationsprogramme und Modellierungsbibliotheken.

Mit dem Simulationsprogramm Snoopy können Modelle auf Basis vieler unterschiedlicher Petri-Netz-Formalismen erstellt und simuliert werden. Biologische Prozesse können mit kontinuierlichen zeitbasierten Petri-Netzen abgebildet werden. Diese Petri-Netze werden von Snoopy unterstützt allerdings wird dieser Formalismus für die Simulation in ein System gewöhnlicher Differentialgleichungen transformiert. Dies führt dazu, dass Markierungen in den Plätzen negativ werden und Transitionen auch rückwärts feuern können, was jeden Petri-Netz-Formalismus ad absurdum führt.

Neben den Vorteilen, die der Cell Illustrator bietet, z.B. die Abbildung komplexer biologischer Prozesse, steht die intransparent Umsetzung des Petri-Netz-Formalismus. Hierdurch ist kein Eingreifen in die Abläufe und keine Erweiterung des Petri-Netz-Formalismus, z.B. um gefärbte Plätze, möglich.

CPN Tools ist das Werkzeug mit dem Kurt Jensen gefärbte Petri-Netze eingeführt hat. Sein Formalismus basiert allerdings auf einem diskreten Zeitkonzept und kann daher nicht für die Simulation zeitbasierter biologischer Prozesse genutzt werden.

Mit der Modellierungsbibliothek BioChem steht zwar eine hervorragende Modelica-Bibliothek für die Modellierung und Simulation von biologischen Prozessen zur Verfügung, aber dieser basiert nicht auf einem Petri-Netz-Formalismus. Dies hat zur Folge, dass die

biologischen Prozesse zu Petri-Netzen abweichend abgebildet werden. Dadurch ist es z.B. möglich, dass unidirektionale Reaktionen dennoch rückwärts ablaufen.

Die Modelica-Bibliothek PNlib ermöglicht es auf Basis des xHPN-Formalismus Modelle zu erzeugen. Hiermit können beispielsweise zeitbasierte biologische Prozesse abgebildet werden. Darüber hinaus lässt sich die Bibliothek auch für eine Vielzahl weiterer Anwendungsgebiete einsetzen. Die Bibliothek PNlib vereinigt einige Vorteile auf sich. So setzt sie den Petri-Netz-Formalismus transparent um und kann kostenfrei genutzt werden. In der Version 1.0 war sie allerdings nicht Modelica-konform und konnte daher nur in Zusammenspiel mit einer bestimmten Modelica-Umgebung verwendet werden.

Der Editor VANESA ist speziell für die Arbeit mit biologischen Netzen ausgelegt. Er ermöglicht zudem die Modellierung mit Petri-Netzen. VANESA kann selbst keine Simulationsberechnungen durchführen, besitzt hierfür allerdings eine Anbindung an die Modelica-Umgebung OpenModelica. Diese kann zusammen mit der Petri-Netz-Bibliothek PNlib eingesetzt werden, um Simulationsergebnisse zu berechnen.

Modelica-Modelle lassen sich als hybride differential-algebraische Gleichungssystem auffassen. Für die Berechnung der Simulationsergebnisse müssen die Modellgleichungen für die unterschiedlichen Problemstellungen der Initialisierung und Simulation aufbereitet und gelöst werden. Für die Lösbarkeit dieser Gleichungssysteme werden individuelle Optimierungen benötigt. Dies war mit dem bisherigen OpenModelica-Backend nur eingeschränkt möglich. PNlib-Modelle können daher nicht berechnet werden und ein Neuentwurf des Backend ist zwingend notwendig.

Die Initialisierung ist für die Simulation von zentraler Bedeutung, da ohne einen konsistenten Anfangszustand die Integration nicht gestartet werden kann. Initialisierungsprobleme lassen sich grundsätzlich in unterbestimmt, überbestimmt, gemischt-bestimmt und regulär unterteilen. Für reguläre Systeme ist die Lösung des Initialisierungsproblems vergleichsweise einfach, da lediglich ein reguläres algebraisches Gleichungssystem gelöst werden muss. Für die Sonderfälle unterbestimmter, überbestimmter und gemischt-bestimmter Systeme wurden in dieser Arbeit algorithmische Lösungsstrategien entwickelt und im OpenModelica-Projekt umgesetzt.

Nach ausgiebiger Analyse der PNlib 1.0 zeigte sich, dass diese eine umfassende Weiterentwicklung benötigte. Dies schließt folgende Punkte ein:

- Die vorhandene Bibliothek Modelica-konform gestalten.
- Die vorhandene Bibliothek auf den Modelica-Sprachstandard 3.3 heben.
- Allgemeine Fehler in der Bibliothek aufspüren und beseitigen.
- Erweiterung der Plätze um Tokenfluss-Informationen vornehmen.

Für die Erweiterung der Petri-Netze um gefärbte Plätze wurde der mathematische Formalismus entsprechend definiert.

Anhand eines Teilmodells der Glykolyse wird die praktische Relevanz der vorangegangenen Arbeiten gezeigt. Hierfür wurde zunächst das metabolische Netzwerk in ein Petri-Netz transformiert und in einem weiteren Schritt um gefärbte Plätze erweitert. Dies ermöglicht, den Weg bestimmter Metaboliten, durch Einfärbung derselben, im Netzwerk zu verfolgen.

7.2 Ausblick

In Zukunft könnte die Transformation von biologischen Netzen zu Petri-Netzen automatisiert werden, sodass der Anwender direkt auf biologischen Netzen arbeiten kann.

Für die Kohlenstoffflussanalyse ist es notwendig, die Zusammensetzung der einzelnen Metaboliten auf atomarer Ebene zu kennen. Diese Informationen müssen bisher händisch eingegeben und könnte durch die Integration einer entsprechenden Datenbank erleichtert werden.

Des Weiteren könnten mit VANESA die biologischen Modelle zusätzlich als Modelica-Modelle für die BioChem-Bibliothek exportiert werden. Wie in den Grundlagen beschrieben, beruht diese Bibliothek auf einem ODE-Ansatz und nicht auf einem Petri-Netz-Formalismus, und setzt so die Simulation mittels eines definierten mathematischen Formalismus um. Aufgrund des unterschiedlichen Ansatzes beider Bibliotheken (PNlib und BioChem) unterscheiden sich im Allgemeinen auch die Simulationsergebnisse. In dem ODE-System sind die Reaktionsrichtungen nicht strikt vorgegeben und Stoffmengen/Konzentrationen können negativ werden. Eine Gegenüberstellung beider Modellierungsansätze wäre so auf Basis bestehender Modelle möglich und der Einfluss des Modellierungsansatzes von biologischen Prozessen auf die Simulationsergebnisse könnte weiter untersucht werden.

Die Integration des OpenModelica-Compilers in VANESA könnte weiter verbessert werden. Derzeit kann VANESA den OpenModelica-Compiler aufrufen, Warnungen (z.B. vom unitChecking-Modul) über den Standardstream und Simulationsergebnisse über eine TCP/IP-Schnittstelle zu empfangen. Besser wäre es, den OpenModelica-Compiler direkt als Bibliothek zu linken und so eine effiziente Integration zu gewährleisten.

Die PNlib ist nicht auf biologische Anwendungen beschränkt und kann allgemein als Petri-Netz-Bibliothek für zeitbasierte Anwendungen genutzt werden. Hierfür könnte die Bibliothek um weitere Konzepte, wie der Faltung von Transitionen und Fuzzy-Konzepten, erweitert werden. Zudem könnte VANESA zu einer allgemeinen Petri-Netz-Oberfläche ausgebaut werden und in einer spezialisierten Form weiterhin für biologische Anwendungen dienen. Die spezialisierte Form könnte durch ein Plugin-System umgesetzt werden. Dies würde erlauben, ohne großen Aufwand weitere spezialisierte Petri-Netz Werkzeuge zu bauen, z.B. für Prozessplanung.

Diese Arbeit deckt den Aspekt der Kohlenstoffflussanalyse mit der PNlib und OpenModelica ab. Darauf sind die Möglichkeiten dieser Bibliothek allerdings nicht beschränkt.

Anhang A: OpenModelica-Backend-Module

Nachfolgend werden alle OpenModelica-Backend-Module[1] beschrieben, die standardmäßig eingeschaltet sind. Zu dem Modul ist zusätzlich das Paket, die Methode und die Phase angegeben, in der sie eingesetzt werden. Die Module sind alphabetisch sortiert.

Modul	**calculateStateSetsJacobians**
Paket	SymbolicJacobian
Methode	calculateStateSetsJacobians
Phase	Simulation
Kurzbeschreibung	Diese Modul berechnet symbolische Jacobimatrizen für die dynamische Zustandswahl.

Modul	**calculateStrongComponentJacobians**
Paket	SymbolicJacobian
Methode	calculateStrongComponentJacobians
Phase	Initialisierung, Simulation
Kurzbeschreibung	Dieses Modul berechnet symbolische Jacobimatrizen für lineare und nichtlineare Systeme. Für nichtlineare Systeme, die benutzer-definierte Funktionen enthalten, werden derzeit keine symbolischen Jacobimatrizen generiert. Mit dem Flag `-d=-NLSanalyticJacobian` lässt sich das Generieren der symbolischen Jacobimatrix für nichtlineare Systeme abschalten.

Modul	**clockPartitioning**
Paket	SynchronousFeatures
Methode	clockPartitioning
Phase	Vorverarbeitung
Kurzbeschreibung	Dieses Modul analysiert das flache Modell auf unabhängige Modellteile. Diese werden dann in separate Datenstrukturen abgelegt, um so im weiteren Transformationsprozess Laufzeitverbesserungen für alle Module zu erlangen, die schlechter als linear zur Systemgröße skalieren. Zusätzlich wird die Partitionierung für die *Synchronous Features* durchgeführt.

[1] Diese Auflistung bezieht sich auf den Stand von Oktober 2016:
⇒ OMCompiler: e2728465b637d72c4dfdb1de96993f0c5908262e

Modul	comSubExp
Paket	CommonSubExpression
Methode	commonSubExpressionReplacement
Phase	Vorverarbeitung
Kurzbeschreibung	Dieses Modul detektiert in einigen speziellen Fällen gemeinsame Teilausdrücke und optimiert diese durch Gausselimination.

Modul	constantLinearSystem
Paket	SymbolicJacobian
Methode	constantLinearSystem
Phase	Simulation
Kurzbeschreibung	Dieses Modul wertet lineare Systeme der Form $A \cdot x = b$ aus, sofern die Matrix A und rechte Seite b des Systems konstant sind.

Modul	detectJacobianSparsePattern
Paket	SymbolicJacobian
Methode	detectSparsePatternODE
Phase	Simulation
Kurzbeschreibung	Dieses Modul berechnet die dünnbesetzte Struktur des ODE-Systems.

Modul	encapsulateWhenConditions
Paket	FindZeroCrossings
Methode	encapsulateWhenConditions
Phase	Vorverarbeitung
Kurzbeschreibung	Dieses Modul führt für jede `when`-Bedingung eine `boolsche` Variable ein.

Modul	evalFunc
Paket	EvaluateFunctions
Methode	evalFunctions
Phase	Vorverarbeitung
Kurzbeschreibung	Dieses Modul wertet konstante Funktionensaufrufe vollständig oder teilweise aus. Einzelne konstante Outputs werden als neue Gleichungen eingeführt.

Modul	evaluateReplaceProtectedFinalEvaluateParameters
Paket	EvaluateParameter
Methode	evaluateReplaceProtectedFinalEvaluateParameters
Phase	Vorverarbeitung
Kurzbeschreibung	Dieses Modul wertet Parameter aus, die nach dem Kompilierungsprozess nicht mehr verändert werden können. Dies sind strukturelle Parameter und solche die als `final` oder `protected` definiert sind.

Modul	expandDerOperator
Paket	BackendDAEOptimize
Methode	expandDerOperator
Phase	Vorverarbeitung
Kurzbeschreibung	Dieses Modul berechnet die Ableitung nach der Zeit für alle Ausdrücke, die innerhalb des `der`-Operators stehen.

Modul	findStateOrder
Paket	IndexReduction
Methode	findStateOrder
Phase	Vorverarbeitung
Kurzbeschreibung	Dieses Modul sammelt Informationen über direkte Zusammenhänge von potentiellen Zuständen und Variablen, die deren Ableitungen beschreiben. Diese Informationen werden später in der Indexreduktion verwendet.

Modul	inlineArrayEqn
Paket	InlineArrayEquations
Methode	inlineArrayEqn
Phase	Vorverarbeitung, Simulation
Kurzbeschreibung	Dieses Modul flacht alle Array-Gleichungen aus. Das bedeutet, dass für jedes Element eine eigene skalare Gleichung angelegt wird.

Modul	inputDerivativesUsed
Paket	SymbolicJacobian
Methode	inputDerivativesUsed
Phase	Simulation
Kurzbeschreibung	Dieses Modul überprüft, ob Ableitungen von sogenannten Toplevel-Input-Variablen im Modell vorkommen. Dies ist nicht zulässig und führt daher zu einer Fehlermeldung.

Modul	lateInlineFunction
Paket	BackendInline
Methode	lateInlineFunction
Phase	Simulation
Kurzbeschreibung	Dieses Modul bettet den Funktionsrumpf von Funktionsaufrufen mit der Annotation `LateInline=true` in das Gleichungssystem ein.

Modul	normalInlineFunction
Paket	normalInlineFunction
Methode	normalInlineFunction
Phase	Vorverarbeitung
Kurzbeschreibung	Dieses Modul ersetzt Funktionsaufrufe, die durch die Inline-Annotation markiert sind, durch ihren Funktionskörper.

Modul	removeConstants
Paket	BackendDAEOptimize
Methode	removeConstants
Phase	Simulation
Kurzbeschreibung	Dieses Modul ersetzt alle Konstanten in dem Gleichungssystem durch ihren Zahlenwert.

Modul	removeEqualFunctionCalls
Paket	BackendDAEOptimize
Methode	removeEqualFunctionCalls
Phase	Vorverarbeitung
Kurzbeschreibung	Dieses Modul substituiert, falls möglich, Funktionsaufrufe durch bestehende Modellvariablen: $a + f(\dots) = 0$ $b + f(\dots) = 0$ ⇨ $f(\dots) = -a$ $b - a = 0$ Das linke System enthält zwei Funktionsaufrufe von $f(\dots)$ und würde durch diese Modul in das rechte System transformiert werden, in dem nur noch ein Funktionsaufruf von $f(\dots)$ enthalten ist.

Modul	removeSimpleEquations
Paket	RemoveSimpleEquations
Methode	removeSimpleEquations
Phase	Vorverarbeitung, Simulation
Kurzbeschreibung	Dieses Modul detektiert einfache Gleichungen und entfernt diese aus dem Gleichungssystem. Einfache Gleichungen sind beispielweise $a \pm b = 0$, $a = \text{not } b$ und $a = 3$.

Modul	replaceEdgeChange
Paket	BackendDAEOptimize
Methode	replaceEdgeChange
Phase	Vorverarbeitung
Kurzbeschreibung	Dieses Modul ersetzt die Standardfunktionen `edge` und `change` durch äquivalente Ausdrücke: • `edge(b) = (b and not pre(b))` • `change(v) = (v <> pre(v))`

Modul	simplifyAllExpressions
Paket	BackendDAEOptimize
Methode	simplifyAllExpressions
Phase	Initialisierung, Simulation
Kurzbeschreibung	Dieses Modul führt Vereinfachungen auf mathematischen Teilausdrücken aus, wie z.B. $0 \cdot a \Rightarrow 0$ und $(1) \cdot (1) \Rightarrow 1$.

Modul	simplifyComplexFunction
Paket	BackendDAEOptimize
Methode	simplifyComplexFunction
Phase	Initialisierung, Simulation
Kurzbeschreibung	Dieses Modul bringt Funktionsaufrufe mit mehreren Rückgabewerten in eine Standardform, um im weiteren Übersetzungsprozess weniger Fälle berücksichtigen zu müssen. Zum Beispiel würde die Gleichung $(a, -b) = f(\dots)$ in die Standardform $(a, tmp) = f(\dots)$ überführt werden. Zusätzlich würden die neue Variable tmp und die triviale Gleichung $tmp = -b$ zu dem Gleichungssystem hinzugefügt werden.

Modul	simplifyIfEquations
Paket	BackendDAEOptimize
Methode	simplifyIfEquations
Phase	Vorverarbeitung
Kurzbeschreibung	Dieses Modul untersucht Gleichungen und Ausdrücke auf unerreichbare if/else-Zweige und entfernt diese. Dies kann zu Vereinfachungen der starken Zusammenhangskomponenten führen.

Modul	simplifysemiLinear
Paket	BackendDAEOptimize
Methode	simplifysemiLinear
Phase	Simulation
Kurzbeschreibung	Dieses Modul ersetzt die Modelica-Standardfunktion `semiLinear(x, positiveSlope, negativeSlope)` durch den folgenden Ausdruck: `if x>=0 then positiveSlope*x else negativeSlope*x`.

Modul	simplifyTimeIndepFuncCalls
Paket	BackendDAEOptimize
Methode	simplifyTimeIndepFuncCalls
Phase	Simulation
Kurzbeschreibung	Dieses Modul vereinfacht Standardfunktionsaufrufe, die nur von einem Parameter p abhängen: • $\mathrm{pre}(p) \Rightarrow p$ • $\mathrm{der}(p) \Rightarrow 0.0$ • $\mathrm{change}(p) \Rightarrow \mathrm{false}$ • $\mathrm{edge}(p) \Rightarrow \mathrm{false}$

Modul	solveSimpleEquations
Paket	ExpressionSolve
Methode	solveSimpleEquations
Phase	Initialisierung, Simulation
Kurzbeschreibung	Dieses Modul löst Gleichungen, falls möglich, die zwar nur von einer Variablen x abhängen, jedoch nicht trivialer Weise nach dieser aufgelöst werden können, wie zum Beispiel $\sin(x) = x + 0{,}1$. Sollte das Modul nicht in der Lage sein, eine solche Gleichung aufzulösen, wird sie als nicht-lineares System mit der Dimension 1 markiert und zur Laufzeit iterativ gelöst.

Modul	tearingSystem
Paket	Tearing
Methode	tearingSystem
Phase	Initialisierung, Simulation
Kurzbeschreibung	Dieses Modul reduziert die Dimension linearer und nichtlinearer Systeme, falls möglich, durch das Aufbrechen algebraischer Schleifen.

Anhang B: Modell „Glykolyse in Saccharomyces cerevisiae“

Das Modell der Glykolyse in Saccharomyces cerevisiae entspricht weitestgehend der Beschreibung aus [23].

Startwerte

Name	Wert	Einheit
ACA	1,48153	$mmol$
ADP	1,5	$mmol$
ATP	2,1	$mmol$
BPG	0,00027	$mmol$
DHAP	2,95	$mmol$
EtOH	19,2379	$mmol$
F6P	0,49	$mmol$
FBP	4,64	$mmol$
G6P	4,2	$mmol$
GAP	0,115	$mmol$
Glc	0,573074	$mmol$
Glcx	1,55307	$mmol$
Glyc	4,196	$mmol$
NAD	0,65	$mmol$
NADH	0,33	$mmol$
PEP	0,04	$mmol$
Pyr	8,7	$mmol$

Reaktion 1: inGlc

$$\rightarrow Glcx$$

$$v_1 = y \cdot k_0 \cdot [Glcx]_0$$

Parameter	**Wert**	**Einheit**
k_0	0,048	$1/min$
y	59,0	1
$[Glcx]_0$	18,5	$mmol$

Reaktion 2: GlcTrans

$$Glcx \underset{v_2^-}{\overset{v_2^+}{\rightleftharpoons}} Glc$$

$$v_2^+ = v_m \frac{\frac{[Glcx]}{k_{Glc}}}{1 + \frac{[Glcx]}{k_{Glc}} + \frac{k_{p2}\frac{[Glcx]}{k_{Glc}}+1}{k_{p2}\frac{[Glc]}{k_{Glc}}+1} \cdot \left(1 + \frac{[Glc]}{k_{Glc}} + \frac{[G6P]}{k_{iG6P}} + [Glc]\frac{[G6P]}{k_{Glc} \cdot k_{iiG6P}}\right)}$$

$$v_2^- = v_m \frac{\frac{[Glc]}{k_{Glc}}}{1 + \frac{[Glc]}{k_{Glc}} + \frac{k_{p2}\frac{[Glc]}{k_{Glc}}+1}{k_{p2}\frac{[Glcx]}{k_{Glc}}+1} \cdot \left(1 + \frac{[Glcx]}{k_{Glc}} + \frac{[G6P]}{k_{iG6P}} + [Glc]\frac{[G6P]}{k_{Glc} \cdot k_{iiG6P}}\right)}$$

Parameter	**Wert**	**Einheit**
v_m	1014,96	$mmol/min$
k_{Glc}	1,7	$mmol$
k_{iG6P}	1,2	$mmol$
k_{p2}	1,0	$mmol$
k_{iiG6P}	7,2	$mmol$

Reaktion 3: HK

$$Glc + ATP \rightarrow G6P + ADP$$

$$v_3 = v_m \frac{[ATP] \cdot [Glc]}{k_{dGlc} \cdot k_{ATP} + k_{Glc} \cdot [ATP] + k_{ATP} \cdot [Glc] + [Glc] \cdot [ATP]}$$

Parameter	**Wert**	**Einheit**
v_m	51,7547	$mmol/min$
k_{ATP}	0,1	$mmol$
k_{Glc}	0,0	$mmol$
k_{dGlc}	0,37	$mmol$

Reaktion 4: PGI

$$G6P \underset{v_4^-}{\overset{v_4^+}{\rightleftharpoons}} F6P$$

$$v_4^+ = v_m \frac{[G6P]}{k_{G6P} + [G6P] + \frac{k_{G6P}}{k_{F6P}} \cdot [F6P]}$$

$$v_4^- = v_m \frac{\frac{[F6P]}{k_{eq}}}{k_{G6P} + [G6P] + \frac{k_{G6P}}{k_{F6P}} \cdot [F6P]}$$

Parameter	**Wert**	**Einheit**
v_m	496,042	$mmol/min$
k_{G6P}	0,8	$mmol$
k_{F6P}	0,15	$mmol$
k_{eq}	0,13	1

Reaktion 5: PFK

$$F6P + ATP \rightarrow FBP + ADP$$

$$v_5 = v_m \frac{[F6P]^2}{k_5 \cdot \left(1 + \kappa \frac{[ATP]^2}{[ADP]^2}\right) + [F6P]^2}$$

Parameter	**Wert**	**Einheit**
v_m	45,4327	$mmol/min$
k_5	0,021	$mmol$
κ	0,15	$mmol$

Reaktion 6: ALD

$$FBP \underset{v_6^-}{\overset{v_6^+}{\rightleftharpoons}} DHAP + GAP$$

$$v_6^+ = v_f \frac{[FBP]}{k_{FBP} + [FBP] + \frac{[GAP]\cdot k_{DHAP}\cdot v_f}{k_{eq}\cdot v_r} + \frac{[DHAP]\cdot k_{GAP}\cdot v_f}{k_{eq}\cdot v_r} + \frac{[FBP]\cdot[GAP]}{k_{iGAP}} + \frac{[GAP]\cdot[DHAP]\cdot v_f}{k_{eq}\cdot v_r}}$$

$$v_6^- = v_f \frac{\frac{[GAP]\cdot[FBP]}{k_{eq}}}{k_{FBP} + [FBP] + \frac{[GAP]\cdot k_{DHAP}\cdot v_f}{k_{eq}\cdot v_r} + \frac{[DHAP]\cdot k_{GAP}\cdot v_f}{k_{eq}\cdot v_r} + \frac{[FBP]\cdot[GAP]}{k_{iGAP}} + \frac{[GAP]\cdot[DHAP]\cdot v_f}{k_{eq}\cdot v_r}}$$

Parameter	**Wert**	**Einheit**
v_f	2207,82	$mmol/min$
v_r	11039,1	$mmol$
k_{eq}	0,081	$mmol$
k_{FBP}	0,3	$mmol$
k_{GAP}	4,0	$mmol$
k_{iGAP}	10,0	$mmol$
k_{DHAP}	2,0	$mmol$

Reaktion 7: TIM

$$DHAP \underset{v_7^-}{\overset{v_7^+}{\rightleftharpoons}} GAP$$

$$v_7^+ = v_m \frac{[DHAP]}{k_{DHAP} + [DHAP] + [GAP] \cdot \frac{k_{DHAP}}{k_{GAP}}}$$

$$v_7^- = v_m \frac{\frac{[GAP]}{k_{eq}}}{k_{DHAP} + [DHAP] + [GAP] \cdot \frac{k_{DHAP}}{k_{GAP}}}$$

Parameter	Wert	Einheit
v_m	116,365	$mmol/min$
k_{GAP}	1,27	$mmol$
k_{DHAP}	1,23	$mmol$
k_{eq}	0,055	1

Reaktion 8: GAPDH

$$GAP + NAD \underset{v_8^-}{\overset{v_8^+}{\rightleftharpoons}} BPG + NADH$$

$$v_8^+ = v_m \frac{[GAP] \cdot [NAD]}{k_{GAP} \cdot k_{NAD} \cdot \left(1 + \frac{[GAP]}{k_{GAP}} + \frac{[BPG]}{k_{BPG}}\right) \cdot \left(1 + \frac{[NAD]}{k_{NAD}} + \frac{[NADH]}{k_{NADH}}\right)}$$

$$v_8^- = v_m \frac{\frac{[BPG] \cdot [NADH]}{k_{eq}}}{k_{GAP} \cdot k_{NAD} \cdot \left(1 + \frac{[GAP]}{k_{GAP}} + \frac{[BPG]}{k_{BPG}}\right) \cdot \left(1 + \frac{[NAD]}{k_{NAD}} + \frac{[NADH]}{k_{NADH}}\right)}$$

Parameter	Wert	Einheit
v_m	883,858	$mmol/min$
k_{GAP}	0,6	$mmol$
k_{BPG}	0,01	$mmol$
k_{NAD}	0,1	$mmol$
k_{NADH}	0,06	$mmol$
k_{eq}	0,0055	1

Reaktion 9: lpPEP

$$BPG + ADP \underset{v_9^-}{\overset{v_9^+}{\rightleftharpoons}} PEP + ATP$$

$$v_9^+ = k_f \cdot [BPG] \cdot [ADP]$$

$$v_9^- = k_r \cdot [PEP] \cdot [ATP]$$

Parameter	Wert	Einheit
k_f	443866,0	$min^{-1} \cdot mmol^{-1}$
k_r	1528,62	$min^{-1} \cdot mmol^{-1}$

Reaktion 10: PK

$$PEP + ADP \rightarrow Pyr + ATP$$

$$v_{10} = v_m \cdot \frac{[PEP] \cdot [ADP]}{(k_{PEP} + [PEP]) \cdot (k_{ADP} + [ADP])}$$

Parameter	Wert	Einheit
v_m	343,096	$mmol/min$
k_{ADP}	0,17	$mmol$
k_{PEP}	0,2	$mmol$

Reaktion 11: PDC

$$Pyr \rightarrow ACA$$

$$v_{11} = v_m \frac{[Pyr]}{(k_{11} + [Pyr])}$$

Parameter	Wert	Einheit
v_m	53,1328	$mmol/min$
k_{11}	0,3	$mmol$

Reaktion 12: ADH

$$ACA + NADH \rightarrow EtOH + NAD$$

$$v_{12} = v_m \frac{[ACA][NADH]}{([NADH] + k_{NADH}) \cdot ([ACA] + k_{ACA})}$$

Parameter	Wert	Einheit
v_m	89,8023	$mmol/min$
k_{NADH}	0,1	$mmol$
k_{ACA}	0,71	$mmol$

Reaktion 15: lpGlyc

$$DHAP + NADH \rightarrow Glyc + NAD$$

$$v_{15} = v_m \frac{[DHAP]}{k_{DHAP} \cdot \left(1 + \frac{k_{iNADH}}{[NADH]} \cdot \left(1 + \frac{[NAD]}{k_{iNAD}}\right)\right) + [DHAP] \cdot \left(1 + \frac{k_{NADH}}{[NADH]} \cdot \left(1 + \frac{[NAD]}{k_{iNAD}}\right)\right)}$$

Parameter	Wert	Einheit
v_m	81,4797	$mmol/min$
k_{NADH}	0,13	$mmol$
k_{DHAP}	25,0	$mmol$
k_{iNADH}	0,034	$mmol$
k_{iNAD}	0,13	$mmol$

Reaktion 23: consum

$$ATP \rightarrow ADP$$

$$v_{23} = k \cdot [ATP]$$

Parameter	**Wert**	**Einheit**
k	3,2076	$1/min$

Reaktion 24: consum

$$NAD \rightarrow NADH$$

$$v_{24} = k \cdot [NAD]$$

Parameter	**Wert**	**Einheit**
k	5,31	$1/min$

Abbildungsverzeichnis

Definitionsverzeichnis

Symbolverzeichnis

$\mathbb{N}$	$= \{1, 2, 3, \dots\}$	Menge der natürlichen Zahlen
$\mathbb{N}_0$	$= \mathbb{N} \cup \{0\}$	Menge der natürlichen Zahlen, einschließlich der Null
$\mathbb{R}^+$	$= \{x \in \mathbb{R} \mid x > 0\}$	Menge der positiven reellen Zahlen
$\mathbb{R}_0^+$	$= \mathbb{R}^+ \cup \{0\}$	Menge der nicht-negativen reellen Zahlen
$\mathbf{0}$	$= (0, \dots, 0)^\top$	Der Nullvektor
$\mathbf{1}$	$= (1, \dots, 1)^\top$	Der Einsvektor
P	$= \{p_1, \dots, p_m\}$	Menge der Plätze
T	$= \{t_1, \dots t_n\}$	Menge der Transitionen
d	$: T \to \mathbb{R}^+$	Verzögerungsfunktion
f	$: F \cup B \to \mathbb{N}$	Gewichtungsfunktion
f	$: (F \cup B, Z_r, time) \to \mathbb{N}_0$	funktionale Gewichtungsfunktion
f	$: (F \cup B, Z_r, time) \to \mathbb{R}_0^+$	(kontinuierliche) funktionale Gewichtungsfunktion
$\boldsymbol{f}$	$: F \cup B \to \mathbb{N}_0^k$	gefärbte Gewichtungsfunktion
$\boldsymbol{f}$	$: (F \cup B, Z_r, time) \to \mathbb{N}_0^k$	gefärbte funktionale Gewichtungsfunktion
$\boldsymbol{f}$	$: (F \cup B, Z_r, time) \to \mathbb{R}_0^{+k}$	(kontinuierliche) gefärbte funktionale Gewichtungsfunktion
$\boldsymbol{z}$	$: P \to \mathbb{N}_0$	Zustand oder Markierung
$\boldsymbol{z}$	$: P \to \mathbb{R}_0$	(kontinuierlicher) Zustand oder Markierung
v	$: T \to \mathbb{R}_0^+$	maximale Geschwindigkeitsfunktion

Literaturverzeichnis

[1] S. Proß, Hybrid Modeling and Optimization of Biological Processes, Bielefeld: Dissertation, Universität Bielefeld, 2013.

[2] Modelica Association, “Modelica and the Modelica Association,” [Online]. Available: https://modelica.org/. [Accessed 4 Juli 2016].

[3] M. Otter, “Modelica Overview,” Modelica Association, 2013.

[4] M. Otter, „Objektorientierte Modellierung Physikalischer Systeme, Teil 3,“ *at – Automatisierungstechnik,* Nr. 48, 2000.

[5] Modelica Association, Modelica® - A Unified Object-Oriented Language for Systems Modeling - Language Specification (Version 3.3 Revision 1), 2014.

[6] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach, 2 Hrsg., Wiley-IEEE Press, 2015.

[7] M. M. Tiller, Introduction to Physical Modeling with Modelica, Springer, 2001.

[8] E. Larsdotter Nilsson und P. Fritzson, „A Metabolic Specialization of a General Purpose Modelica Library for Biological and Biochemical Systems,“ in *Proceedings of the 4th International Modelica Conference*, G. Schmitz, Hrsg., Hamburg, Germany, 2005.

[9] Open Source Modelica Consortium, OpenModelica User's Guide.

[10] J. Frenkel, Entwicklung eines Modelica Compiler BackEnds für große Modelle, Dresden: Dissertation, Technischen Universität Dresden, 2014.

[11] K. Brenan, S. Campbell und L. Petzold, Numerical Solution of Initial–Value Problems in Differential–Algebraic Equations, Elsevier Science Publishers, 1996.

[12] C. Pantelides, „The Consistent Initialization of Differential-algebraic Systems,“ *SIAMJournalon Scientific and Statistical Computing,* 1988.

[13] S. Mattsson und G. Söderlind, „Index Reduction in Differential-algebraic Equations Using Dummy Derivatives,“ *SIAM Journal on Scientific Computing,* 1993.

[14] E. Carpanzano, „Order Reduction of General Nonlinear DAE Systems by Automatic Tearing,“ *Mathematical and Computer Modelling of Dynamical Systems,* Bd. 6, Nr. 2, pp. 145-168, 2000.

[15] C. A. Petri, Kommunikation mit Automaten, Bonn: Dissertation, Technische Hochschule Darmstadt, 1962.

[16] W. Reisig, Petrinetze - Modellierungstechnik, Analysemethoden, Fallstudien, Vieweg+Teubner Verlag, 2010.

[17] T. Lask, H.-J. Kruse und B. Bachmann, „Simulation und Optimierung der Personalplanung im Pflegebereich von Krankenhäusern durch Petri-Netz-Modelle," in *Angewandte mathematische Modellierung und Optimierung - Ausgewählte Modelle, Methoden, Fallstudien*, Bielefeld, Fachhochschule Bielefeld, 2016, pp. 36-61.

[18] K. Jensen und L. M. Kristensen, Colored Petri Nets, Springer Berlin Heidelberg, 2009.

[19] B. Bachmann, T. Kleine-Döpke, H.-J. Kruse, L. Ochel und S. Proß, Petri-Netz-Formalismen und Lösungsansätze für allgemeine Konfliktsituationen bei Feuerprozessen in Petri-Netz-Modellen, Bielefeld: Fachhochschule Bielefeld, 2014.

[20] E. Klipp, W. Liebermeister, C. Wierling, A. Kowald, H. Lehrbach and R. Herwig, Systems Biology: A Textbook, Weinheim: WILEY-VCH Verlag GmbH & Co. KGaA, 2009.

[21] A. F. Holleman, E. Wiberg und N. Wiberg, Lehrbuch der Anorganischen Chemie, 102 Hrsg., Berlin: De Gruyter, 2007.

[22] G. E. Briggs und J. B. S. Haldane, „A Note on the Kinetics of Enzyme Action," *Biochemical Journal,* pp. 338-339, 1925.

[23] F. Hynne, S. Danø and P. G. Sørensen, "Full-scale model of glycolysis in Saccharomyces cerevisiae," *Biophysical Chemistry,* no. 94, pp. 121-163, 2001.

[24] E. Larsdotter Nilsson und F. Peter, „BioChem - A Biological and Chemical Library for Modelica," in *Proceedings of the 3nd International Modelica Conference*, Linköping, Sweden, 2003.

[25] J. Brugård, D. Hedberg, M. Cascante, G. Cedersund, A. Gómez-Garrido, D. Maier, E. Nyman, V. Selivanov and P. Strålfors, "Creating a Bridge between Modelica and the Systems Biology Community," in *Proceedings of the 7th International Modelica Conference*, F. Casella, Ed., Linköping University Electronic Press, 2009.

[26] S. Proß and B. Bachmann, "PNlib - An Advanced Petri Net Library for Hybrid Process Modeling," in *Proceedings of the 9th International Modelica Conference*, M. Otter and D. Zimmer, Eds., Munich, Germany, Linköping University Electronic Press, 2012.

[27] C. Brinkrolf, S. J. Janowski, B. Kormeier, M. Lewinski, K. Hippe, D. Borck und R. Hofestädt, „VANESA - A Software Application for the Visualization and Analysis of Networks in Systems Biology Applications," *Journal of Integrative Bioinformatics,* 2014.

[28] K. Hippe, B. Kormeier, S. Janowski, T. Töpel und R. Hofestädt, „DAWIS-M.D. 2.0 - A Data Warehouse Information System for Metabolic Data," *Proceedings of the 7th International Symposium on Integrative Bioinformatics,* 2011.

[29] M. Heiner, M. Herajy, F. Liu, C. Rohr and M. Schwarick, "Snoopy - A Unifying Petri Net Tool," in *Application and Theory of Petri Nets*, S. Haddad and L. Pomello, Eds., Hamburg, Germany, Springer-Verlag, 2012, pp. 398-407.

[30] H. Alla und R. David, „Continous and hybrid Petri nets," *Journal of Circuits, Systems and Computers,* pp. 159-188, Februar 1998.

[31] M. Nagasaki, A. Doi, H. Matsuno und S. Miyano, „Genomic Object Net: I. A platform for modelling and simulating biopathways.," *Applied bioinformatics,* 2003.

[32] M. Nagasaki, A. Saito, E. Jeong, C. Li, K. Kojima, E. Ikeda und S. Miyano, „Cell illustrator 4.0: a computational platform for systems biology," *Studies in health technology and informatics,* 2011.

[33] H. Matsuno, Y. Tanaka, H. Aoshima, A. Doi, M. Matsui und S. Miyano, „Biopathways Representation and Simulation on Hybrid Functional Petri Net," *In Silico Biology,* Bd. 3, p. 389–404, 2003.

[34] A. Doi, S. Fujita, H. Matsuno, M. Nagasaki und S. Miyano, „Constructing Biological Pathway Models with Hybrid Functional Petri Nets," *In Silico Biology,* Bd. 4, p. 271–291, 2004.

[35] K. Jensen und L. M. Kristensen, „Formal Definition of Non-hierarchical Coloured Petri Nets," in *Coloured Petri Nets*, Springer, 2009, pp. 79-94.

[36] K. Jensen und L. M. Kristensen, „Formal Definition of Timed Coloured Petri Nets," in *Coloured Petri Nets*, Springer, 2009, pp. 257-272.

[37] F. Casella, „Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives," *Proceedings of the 11th International Modelica Conference,* pp. 459-468, September 2015.

[38] B. Bachmann, P. Aronsson and P. Fritzson, "Robust Initialization of Differential-algebraic Equations," *Proceedings 5th International Modelica Conference,* pp. 607-614, September 2006.

[39] I. S. Duff, K. Kaya und B. Uçcar, „Design, Implementation, and Analysis of Maximum Transversal Algorithms,“ *ACM Transactions on Mathematical Software,* pp. 13:1-13:31, Januar 2012.

[40] L. Ochel, B. Bachmann and F. Casella, "Symbolic Initialization of Over-determined Higher-index Models," *Proceedings 10th International Modelica Conference,* pp. 1179-1187, March 2014.

[41] L. Ochel and B. Bachmann, "Initialization of Equation-based Hybrid Models within OpenModelica," *EOOLT 2013 Proceedings,* April 2013.

[42] M. Kohlstedt, J. Becker und C. Wittmann, „Metabolic fluxes and beyond - systems biology understanding and engineering of microbial metabolism,“ *Applied Microbiology and Biotechnology,* Bd. 88, Nr. 5, pp. 1065-1075, 2010.