Rule-based Simulation of biochemical Reaction Processes

Regelbasierte Simulation von biochemischen Reaktionsprozessen
Master Thesis submitted by
Sebastian Ehmes
November 5, 2018

TECHNISCHE UNIVERSITÄT DARMSTADT

Real-Time Systems Lab

Department of Electrical Engineering and Information Technology (FB18)

Adjunct Member Department of Computer Science (FB20)

Prof. Dr. rer. nat. A. Schürr
Merckstraße 25
64283 Darmstadt

www.es.tu-darmstadt.de

Supervisor:	Prof. Dr. rer. nat. A. Schürr
Advisor:	Lars Fritsche, M.Sc.

ES-M-0131

Declaration of Authorship for the Master Thesis

I warrant that the Master Thesis presented here is my original work, and that I did not receive any external assistance. All references and other sources I used have been appropriately acknowledged. I further declare that the work has not been submitted for the purpose of academic examination anywhere else, either in its original or similar form.

I hereby grant the Real-Time Systems Lab the right to publish, reproduce, and distribute my work.

Darmstadt, November 5, 2018

(S. Ehmes)

Abstract

This masters thesis presents a framework for performing rule-based stochastic simulations of biochemical processes, using existing general-purpose incremental pattern matching tools, such as Viatra and Democles. Domain-specific tools for rule-based simulations are usually highly optimized but lack expressiveness. General-purpose tools, on the other hand, lack in performance but have the advantage of higher expressiveness. The newly developed hybrid pattern matching approach, presented in this thesis, aims to mitigate performance issues of these general-purpose tools, while retaining their advantages. The performance of the framework is evaluated, with regard to speed and memory requirements, using both pattern matching tools and the new hybrid approach. Furthermore, the framework's performance is compared to that of the domain-specific tool KaSim, using the EGF signal pathway model, since it is a popular and well-known example in biochemistry.

Contents

List of Figures

1 Introduction

As of today, many biochemical processes in biological lifeforms are still not completely understood on a molecular level. Most often these processes, such as signal pathways, play a major role in regulating cellular activity, among other things. Diseases like cancer or autoimmune disorders, are examples, where some regulatory aspect in human cells fails to work as intended. In these cases, it is imperative to have a good understanding of the underlying biochemical processes in order to find a treatment. For this purpose, said processes are often simulated, to either study certain aspects more closely or to test the effects of potential treatments.
Over the course of decades many different simulation approaches have been developed in order to further the understanding of biochemical processes. The most prominent among them is the simulation of molecular reactions through the use of systems of ordinary differential equations. This approach requires a deep domain-specific knowledge in order to model processes, by formulating said equation systems. Rule-based simulation, on the other hand, represents a different and more recent approach, where reactions and molecules are described on a higher level of abstraction, which makes it more accessible. Additionally, it requires less effort to model complex processes, such as the initially mentioned signal pathways.
Tools employed in rule-based simulations of biochemical processes are either highly specialized or developed completely independent from certain application domains. Those representing the former category, like Kappa [DFFK07] and its proprietary simulation tool KaSim[1], provide highly efficient and fast simulations for biochemicals processes. However, they grant little to no support for additional calculations of more complex rule preconditions. Tools representing the latter category, like Democles[2] [VD13] or Viatra[3] [VBH+16], are more expressive, which makes it possible to add additional features, such as abstract constraints for rule applications. On the other hand, they are not optimized for specific application domains.

1.1 Goals of the Thesis

The goal of this thesis is to implement a framework for performing rule-based simulations of biochemicals processes. Furthermore, the pattern detection required for such a rule-based approach, will be performed by general-purpose pattern matching tools. As opposed to most existing simulation frameworks, which often employ custom tailored and often less expressive solutions for this purpose. Since the process of pattern detection is often the most runtime expensive part of an application, the employed approach to pattern matching must be chosen carefully.
The framework presented in this thesis, will use incremental pattern matching tools, since they are well suited to react efficiently, when changes within a large model occur frequently, as is the case in biochemical simulations. For this purpose, Democles and Viatra are employed and systematically evaluated. Both tools will be used to investigate

[1] Kappa/KaSim project page: https://kappalanguage.org
[2] Democles Git-Repository and Documentation: https://github.com/eMoflon/emoflon-ibex-democles
[3] Viatra project page: https://www.eclipse.org/viatra/

the effects of various parameters, e.g., model size, on runtime and memory requirements. Additionally, a domain-specific language will be developed, which will provide the means to model entities, rules and initial conditions for rule-based simulations.
Furthermore, a well-known problem will be examined, which occurs when a pattern consists of disjunct sub-patterns. A class of patterns that appears quite frequently in the biochemistry problem domain. In such cases, the amount of possible pattern occurrences may grow exponentially and render the process of pattern matching impossible to execute in a high-performing fashion. Hence, this thesis aims to develop a new method to calculate the number of pattern occurrences, by separately evaluating their constituent non-disjunct sub-patterns and reassemble them on demand.

1.2 Thesis Structure

The following section 2 contains the necessary basics that help to understand details of the implementation and the given examples. For this reason, subsection 2.1 introduces basic terms of biochemistry and presents the EGF signal pathway, an example of a biochemical process (subsection 2.1.1). Subsection 2.2 introduces the basics of rule-based modeling, within the context of biochemistry, and explains stochastic simulation for biochemical processes (subsection 2.2.1), the basis of most rule-based simulations. As the last part of subsection 2.2.2, the Kappa modeling language is presented, which can be used to model biochemical processes. Subsection 2.3 explains the concept of model-driven software engineering, since this thesis relies on its basic concepts, such as the use and specification of domain-specific languages (subsection 2.3.1), model transformation through rule application (subsection 2.3.2), and pattern matching (subsection 2.3.3).

Section 3 presents the implementation and important components of the simulation framework developed in this thesis. Subsection 3.1 gives a brief overview of the framework. Subsection 3.2 presents a new domain-specific language, inspired by Kappa, for modeling biochemical processes, which was developed for the framework in this thesis. Subsection 3.3 presents the core components of the framework, which implement a rule-based stochastic simulation. Since pattern matching is indispensable for rule application, subsection 3.4 explains how the general-purpose pattern matching tools - Viatra and Democles - were integrated. Furthermore, hybrid pattern matching, a new approach to tackle performance problems caused by disjunct patterns in the biochemical context, is presented (subsection 3.4.4).

Section 4 presents and discusses simulation results of two different biochemical process models (subsection 4.1). Subsequently, in subsection 4.2, simulation runtime measurements, with and without using hybrid pattern matching, are presented. Furthermore, the performance of the two pattern matching tools is discussed as well as possible gains, when hybrid pattern matching is activated. In subsection 4.3, runtime measurements of the so-called EGF signal pathway[CC79], an example of a biochemical process from active research, are presented and compared to those of the domain-specific simulation tool KaSim.

Section 5 presents related works that also discussed topics centered around rule-based modeling and simulation of biochemical processes. Additionally, the KaSim tool will be presented as well.

At the end, section 6 recapitulates the different achievements of this thesis, briefly discusses the results of the evaluation section and presents possible future improvements.

2 Theoretical Background

The main goal of this thesis is to create a framework for rule-based simulations of biochemical reactions, which can be used to investigate the nature of certain biochemical processes. In order to help understand the given examples and details of the implementation, subsection 2.1 will explain terms from the field of biochemistry that appear in this thesis. Additionally, a running example modeling a signal pathway is given, which will be used later to evaluate performance and illustrate one of many use cases. Section 2.2 summarizes simulation techniques for biochemical processes, shows the concept of rule-based modeling in general and explains the mechanics of stochastic simulation. Furthermore, *Kappa* is introduced, a modeling language used to model biochemical signal networks. Section 2.3 will explain the concepts of model-based software engineering and the purpose of domain-specific languages, herein used for modeling. The presented simulation framework makes use of pattern matching and model transformations. For this reason, the last two subsections are dedicated to these respective topics.

2.1 Biochemical Reactions

Humans, animals, insects, etc. are composed of an incomprehensible amount of cells, which represent the basic structural and functional building blocks of all complex living organisms. A typical animal cell contains cytoplasm enclosed within a membrane that separates the cells interior from its surroundings. Cytoplasm is comprised of the organelles, i.e., the internal substructures of the cell, the nucleus and cytosol, which is a gel-like transparent substance.
Biomolecules such as proteins, carbohydrates, lipids and nucleic acids are the most prevalent components in cytoplasm. They are the main building blocks for the organelles as well as the nucleus, and represent the actors for all biochemical processes that occur in- and outside of a cell. The main goal of biochemistry is to study and understand the biochemical processes, i.e., the interaction between the previously mentioned actors, that occur in living organisms. Such a process describes the transformation of a chemical compound into a different chemical compound. For example, a protein could connect to another protein and form a larger molecule with different properties.
Receptors on the cell membrane enable cells to react to external stimuli and to communicate with each other. Molecules, such as hormones, neurotransmitters or growth factors, coming from outside the cell may activate these receptors for signaling. Once activated, a cascade of biochemical reactions is triggered, which in turn activates certain mechanisms inside the cell. These cascades are called *signal pathways* and play a major role in controlling cell growth, activity, division and death. Therefore, it makes sense to simulate these pathways to gain a deeper insight into their inner workings.

An example of one of the above mentioned molecules that can activate a receptor on the cell membrane is the **E**pidermal **G**rowth **F**actor (*EGF*). Stimulating the EGF-**R**eceptor (*EGFR*) triggers the EGF signal pathway, which plays an important role in regulating events in mammalian cells, such as their growth and differentiation, i.e., determination of the cells type [CC79]. Since this pathway is central to cell regulation, it has long

been subject to research concerning cancer and the treatment thereof. Mutated versions of the EGF-Receptor have been found in a plethora of cancer cell types. For example, lung cancer cells [LBS+04], brain tumors [LNR+85], breast cancer cells [CHA+18] and many more. Overexpression of the EGFR caused by mutations has been found to be an important factor in tumor growth. For this reason, new cancer treatments aim to target mutated EGFRs with special antibodies [KWB01]. Compared to conventional proven treatments, like surgery, radiation therapy and chemotherapy, this promising approach is less invasive. Therefore, it causes less damage to healthy cells.
As a matter of fact, treatments targeting the EGFR are still ongoing research, since the role of the EGFR signaling network in regulating mammalians cells is still not completely understood [DFF+07]. For this reason, simulating the EGF pathway is not only an interesting example, but also relevant to ongoing research concerned with the development of new therapies for cancer treatment. As stated at the beginning, the EGF pathway will later be used to evaluate the simulation framework presented in this thesis. Therefore, the next chapter will give an overview of the pathway's intermediate steps and explain the necessary details.

2.1.1 EGF Signal Pathway

The EGF signal pathway is a complex series of 211 reactions and 202 proteins [OMFK05], which in its entirety would be too complicated of an example. Fortunately, Danos et al. [DFF+07] simplified the pathway model in order to create a feasible simulation. In addition to that, a preceding bachelor's thesis, written at the Real-Time Systems Lab, also researched rule-based simulation of biochemical processes. In the mentioned bachelor's thesis the EGF pathway was summarized and used as an example to test the presented simulation tool. This thesis draws from both sources in order to summarize the EGF signal pathway in such a manner, that is digestible for readers coming from a scientific backgrounds other than biochemistry.

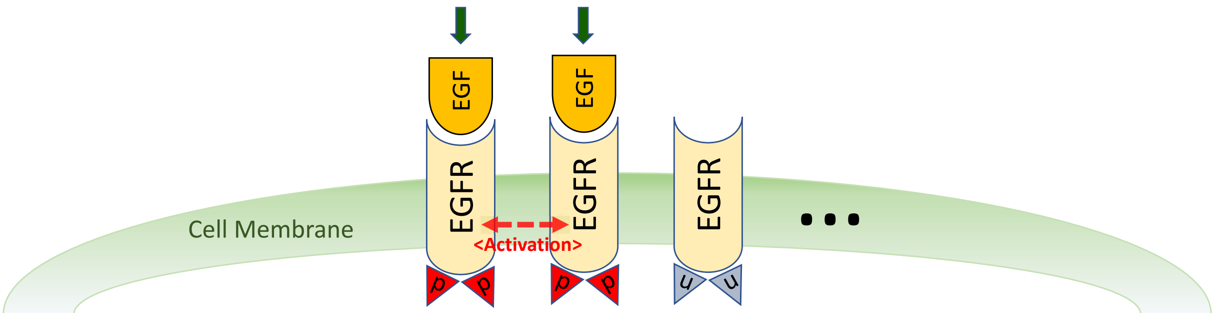


Figure 2.1: EGF Signal Pathway (1)

The process begins when an EGF protein arrives at the cell membrane and binds itself to the receptor protein EGFR, as shown in figure 2.1. After that, the EGFR can bind to a neighboring EGFR that is also bound to an EGF molecule. Both EGFR proteins are now capable of cross-activating each other, which is illustrated by the double arrow in figure 2.1. Cross-activation implies, certain parts of the EGFR residing inside the cell become phosphorylated. That means they undergo phosphorylation, the reversible biochemical process of attaching a phosphoryl group to a biomolecule. Parts that underwent this process are also called phosphorylated residues (red triangles in figure 2.1) and can serve as binding sites for proteins from within the cytoplasm. Unphosphorylated residues (gray triangles in figure 2.1), on the other hand, are unable to bind other proteins.

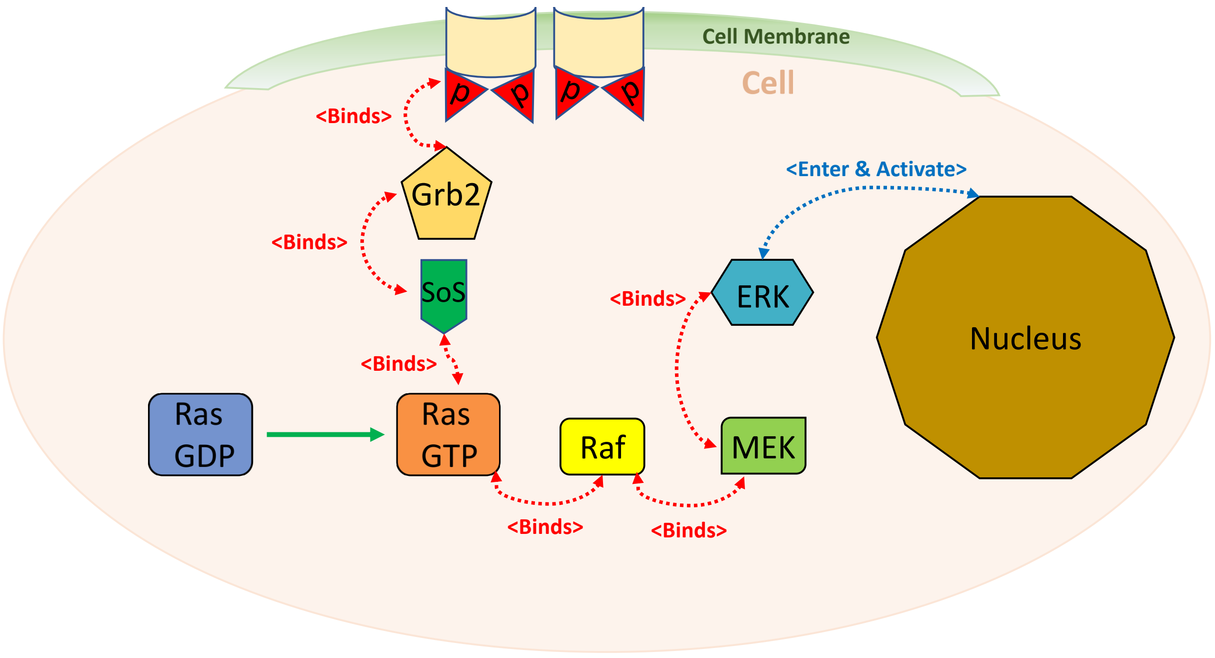


Figure 2.2: EGF Signal Pathway (2)

Inside the cell, an EGRF's phosphorylated residue binds a Grb2 protein, which in turn binds a SoS protein (see figure 2.2). A Ras protein from inside the cell can now bind the SoS protein. As a consequence, it can trade it's low energy GDP for a high energy GTP molecule, as illustrated in 2.2. In the next step, only a RasGTP protein can bind a Raf protein. A bound Raf molecule then becomes activated and binds a MEK protein, which becomes activated as well. After that, an activated MEK protein binds and activates an ERK protein. Finally, an activated ERK protein enters the nucleus and triggers the reproduction of genetic material.
A singular successful completion of the EGF pathway will not lead to unlimited reproduction of the genetic material inside the nucleus. Instead, there are mechanisms which regulate this process. For instance, the deactivation of Ras, Raf, MEK and ERK proteins can interrupt the signal pathway. Additionally, the Ras molecule can bind only one of the following molecules at a time: RasGTP, SoS or Raf. Only the SoS protein is able to bind an inactive Ras, the other molecules have to compete for an active Ras. That means, a RasGTP protein could, for example, bind a RasGTP protein, which in turn prevents it from binding a Raf protein. As a consequence, the signal pathway is halted until a free RasGTP protein can bind a Raf protein.

2.2 Rule-based Modeling in Biochemistry

As mentioned in the previous subsection, signal pathways play a vital role in regulating cell activity. For this reason, a malfunctioning pathway is often the culprit or at least an indicator for cell misbehavior, e.g., uncontrolled growth in case of cancer. Additionally, many biochemical reactions and for that matter signal pathways especially, are very complex, with respect to the number of involved molecules and the number of subsequent reactions needed to complete a signal pathway. Furthermore, observing biochemical processes in detail and in real-time is either not yet possible or very tedious and consequently time consuming and expensive. Hence, signal pathways are often not completely understood and subject to active research. For these reasons, computer simulation has become very common and an effective means to further investigate biochemical processes and, for example, test new drugs or other therapeutic means, before applying them to a living test subject.

The basic question all simulations of biochemical reaction processes try to answer is: Given a number of reaction channels, how does an initial number of molecules, of a certain type, change over time? To answer this, several assumptions are typically made. First and foremost, the simulation has to take place in a fixed volume V, which contains a spatially uniform mixture of the N chemical species, i.e., the involved molecule types. Furthermore, these molecule types can interact through M specified chemical reactions.
The traditional way of simulating a set of biochemical reactions is to use a system of **O**rdinary **D**ifferential **E**quations (*ODE*). According to Gillespie [Gil77], a set of ODEs is constructed under the assumption that the number of molecules of the ith species, contained in the given volume V at time t, can be expressed through a function $F_i(t)$, with $i \in \mathbb{N}$. Additionally, each of the M reactions are assumed to be processes with a continuous rate. Given this, a set of coupled first-order differential equations can be constructed:

$$\begin{aligned} dF_1/dt &= g_1(F_1, ..., F_N) \\ dF_2/dt &= g_2(F_1, ..., F_N) \\ &... \\ dF_N/dt &= g_N(F_1, ..., F_N) \end{aligned} \tag{1}$$

Functions $g_1, ..., g_N$ are determined through properties of the M reactions, e.g., their structure and rate constants. To answer the formerly posed question, one has to solve the so-called reaction rate equations (equation 1) for functions $F_1(t), ..., F_N(t)$, given the above described initial conditions. Solutions to these equations are usually found numerically, through the use of computers, because analytical solutions can only be found for very simple systems.
Despite the great importance and usefulness of this approach, simulation of biochemical reactions through chemical kinetics has several issues. This approach assumes the chemical reaction system to be both continuous and deterministic. Gillespie [Gil77] noticed that this does not reflect real world properties of such processes, which show that molecular population levels can only change in discrete integer amounts. Additionally,

he states that time evolution of a biochemical system is not a deterministic process, for it is impossible to know the exact positions and velocities of molecules, which are properties needed to precisely predict exact molecular population levels. Another problem of simulations through ODEs is the combinatorial nature of signaling pathways. Davos et al. [DFF+07] states that this property leads to an exponential increase in the amount of equations required to model such chemical reactions networks. Within the context of biochemistry, most of the molecular species are proteins that posses multiple sites at which they can be modified, e.g., through phosphorylation. Each site increases the number of states a molecule may have exponentially, which means that 5 modifiable sites will lead to 32 states. The description of a complex consisting of two molecules, with 32 states each, would require 1024 equations. The reason for this, is the fact that every molecular species has its own concentration variable and equation describing its rate of change, which must be known beforehand in order to formulate the ODEs. As a consequence, complex biochemical reaction processes, like signal pathways, can not possibly be formulated as a system of equations and therefore not simulated using this traditional approach.

In combination, these problems have led to the adoption of *rule-based modeling*, which is an attempt at describing biochemical reaction processes in a different fashion. In contrast to the approach of using chemical kinetics and a system of ODEs, rule-based models use a set of rules to define biochemical systems.

$$2H_2 + O_2 \rightarrow 2H_2O \tag{2}$$

The idea is to formulate these rules using the notation of simple chemical reactions describing local events, e.g., equation (2) showing oxyhydrogen. Within the context of rule-based models, molecules are described as *agents*, whose possible interactions are defined by rules that specify how a local pattern of *sites* and their *states* is to be rewritten [DFF+07]. The difference between classic chemical reactions and rule applications in rule-based modeling is that post-translational modifications are not seen as chemical transformations, but as state changes of the involved agents. That means, a protein, despite having been phosphorylated, is still seen as the same entity, but in a different state. Furthermore, in rules only the participating agents and some of their respective sites, i.e., those that are of importance to this rule, are mentioned and not all agents and sites a molecule may have. This essentially avoids the scalability problem that the ODE-based chemical kinetics modeling approaches have. Instead of exhaustively enumerating reactions between fully-specified molecule types, rules only explicitly state those aspects of their involved agents that are actually relevant to the interactions described by the rules.
Rule-based modeling has been successfully implemented in several software tools such as *BioNetGen* (see 5.1), *RuleMonkey* (see 5.2) and Kappa. The latter will be used to illustrate rule-based modeling further and is discussed in subsection 2.2.2.

2.2.1 Stochastic Simulation

Gillespie's algorithm [Gil77] describes the stochastic simulation of coupled chemical reactions. Most approaches that implement rule-based simulations of biochemical processes

are based on the principle of Gillespie's work, as is the case with the framework presented in this thesis. Therefore, its basics, assumptions and procedure will be discussed in this subsection.
Opposed to the previously mentioned ODE-based approach, this algorithm does not assume the chemically reacting system to be continuous and deterministic, since Gillespie argues that this does not reflect reality very well. For example, he states that reaction-rate equations lack the ability to describe fluctuations in molecular population levels. For this and other reasons [Gil77], a new algorithm was developed that takes explicit account of the fact that time evolution in spatially homogeneous chemical systems is a discrete stochastic process. Therefore, Gillespie named his new computational method *stochastic simulation algorithm.*

The basic assumption is that the system is in thermal equilibrium. Hence, molecules will at all times be distributed randomly and uniformly throughout the containing volume V. Gillespie shows that we are not able to calculate the number of all reactions occurring in V. Instead, we can calculate the probability of a reaction occurring in V in any infinitesimal interval. For this reason, it is more appropriate to describe a system of molecules in thermal equilibrium through a reaction probability per unit in time, instead of a reaction rate. Hence, these reactions describe a stochastic Markov process. An important component of this concept is the stochastic reaction constant c_μ, which represents the average probability that a particular set of molecules will react. Generally speaking, the assumption is that each reaction R_μ has its own constant c_μ, with $\mu \in [1, M]$. Additionally, F_i molecules of molecule types $X_i (i \in [1, N])$ exist in a spatially homogeneous mixture inside a volume V. Lastly, these N molecule types can react through the M reaction channels.

$$R_1 : \qquad X_1 + X_2 \longrightarrow X_3 \tag{3}$$

For example, the reaction in equation 3 defines two different molecules, which react and form a third kind of molecule. Its reaction probability can be described as follows:

$$P_{R_1} = F_1 F_2 c_1 dt \tag{4}$$

P_{R_1} in equation 4 describes the probability of a reaction R_1 occurring somewhere in V, within the next infinitesimal time step dt. This probability P_{R_1} depends on the physical properties of the involved molecules and the temperature of the system, described by $c_1 dt$, which is the average probability of the molecules reacting in the next time step dt. On the other hand, P_{R_1} depends on the number of distinct molecule combinations $F_1 F_2$, which can be found in V, with F_1 and F_2 each representing population numbers of species X_1 and X_2.
The calculation of the stochastic time evolution of a spatially homogeneous mixture, containing N molecular species reacting in M reaction channels, is based on the reaction probability density function $P(\tau, \mu)$.

$$P(\tau, \mu) = \begin{cases} a_\mu e^{(-a_0 \tau)} & if: \ 0 \le \tau < \infty \ and \ \mu \in [1, ..., M] \\ 0 & otherwise \end{cases} \tag{5}$$

Given a system at time t that is in the state $(F_1, ..., F_N)$, function $P(\tau,\mu)dt$ expresses the probability for the next occurring reaction in V, happening in the time interval $[t+\tau, t+\tau+d\tau]$ and being a R_μ reaction. In short, it tells us when the next reaction occurs and what kind of reaction it will be. Equation 5 shows a joint probability density function, with the continuous variable τ representing the time step and the discrete variable μ denoting the reaction.

$$a_\mu = h_\mu c_\mu \,, \qquad \mu \in [1, ..., M] \tag{6}$$

Function a_μ expresses the probability of a reaction R_μ occurring in V, given the systems state $(F_1, ..., F_N)$ at time t. Function h_μ, used in equation 6, represents the current state of the system. More precisely, it is the number of distinct molecular combinations in R_μ available in the state at time t. Given the example reaction in equation 3, h_1 of R_1 would have the form $h_1 = F_1 F_2$.

$$a_0 = \sum_{v=1}^{M} a_v = \sum_{v=1}^{M} h_v c_v \tag{7}$$

Equation 5 also shows a function a_0, which is the sum over all a_μ and expresses the probability of any reaction occurring in V, given the system's state. It can also be seen as the system's overall activity.
In order to find a random reaction and the corresponding time interval, we need to generate a random pair (τ, μ), according to the probability density function in equation 5. Gillespie does that by first finding two uniformly distributed random numbers r_1 and r_2, lying within the interval $[0, 1]$.

$$\tau = \left(\frac{1}{a_0}\right) ln\left(\frac{1}{r_1}\right) \tag{8}$$

Given r_1, a time interval τ can be calculated through equation 8, which generates the random number τ according to the probability density function $P_1(\tau) = a_0 e^{-a_0 \tau}$.

$$\sum_{v=1}^{\mu-1} a_v < r_2 a_0 \leq \sum_{v=1}^{\mu} a_v \tag{9}$$

Given r_2, μ representing the index of the selected reaction is determined by satisfying the constraint in equation (9). This produces a random number according to the probability density function $P_2(\mu) = \frac{a_\mu}{a_0}$. Gillespie shows that $P_1(\tau)P_2(\mu) = P(\tau, \mu)$ [Gil77]. Therefore, a valid pair (τ, μ) is produced, which satisfies the density function in equation 5.

The algorithm devised by Gillespie, which simulates the stochastic time evolution of a chemical reacting system, is comprised of 4 steps.

- **Step 1** - In the first step, all variables are initialized. That means M reaction constants c_μ are set, N initial molecular population numbers F_i are given and the time variable t is set to zero.

- **Step 2** - Next, all values for $h_1, ..., h_M$ are calculated, based on current molecular population levels. Using $h_1, ..., h_M$, values for $a_1, ..., a_M$ are assigned according to equation 6. Consequently, a_0 is calculated by summing over $a_1, ..., a_M$, using equation 7.
- **Step 3** - In this step, two random numbers r_1 and r_2 are created, which are then used to calculate τ and μ according to equation 8 and 9.
- **Step 4** - Finally, R_μ is selected, using μ from step 3, and time t is increased by τ. Furthermore, molecular population levels are adjusted to reflect an occurrence of reaction R_μ. Looking at the example in equation 3, reaction R_1 would decrease the population levels F_1 and F_2 of X_1 and X_2 by 1, and increase the population level F_3 of X_3 by 1.

The algorithm will repeat steps 2 through 4 until some predefined termination criterion is fulfilled. Gillespie suggests using a maximum time t, a maximum number of reactions n or a_0 reaching 0, as a trigger to halt the simulation.

2.2.2 Kappa

Modeling biochemical reactions and its constituents in order to simulate complex biochemical processes plays a major role in this thesis. Kappa (κ), a widely used modeling language for biomolecular interactions, served as an inspiration as well as a reference for the design of a new domain-specific language, presented in subsection 3.2 of this thesis. Therefore, this subsection will present Kappa's key features.

Kappa is a domain-specific language used to model proteins and the interactions between them [DL04]. As stated in subsection 2.2, it is an example of the implementation of rule-based modeling. Hence, Kappa does not require the user to write down differential equations to model biochemical processes. Instead, the κ language uses rules to define interactions among molecules.
The main component of a Kappa model is a collection of agents and rules. An agent represents a biomolecular species, i.e., a certain type of biomolecule or complex of biomolecules, that can interreact with other agents. Each agent has a name and a set of sites. The purpose of sites is to serve as an interface, where agents may be bound to other agents. Additionally, sites can have a set of possible internal states. Such a state could potentially be used to describe a phosphorylated or unphosphorylated site of a protein.

Listing 2.1: Agent Declaration

```
1 %agent: A(x,y)                    // Declaration of agent A
2 %agent: B(z{p, q})                // Declaration of agent B
```

The Kappa language makes it fairly comfortable to define agents and sites. The example in listing 2.1 line 1 demonstrates the definition of an agent with name `A`, possessing two sites `x` and `y`. In line 2, an additional agent with name `B` is defined. In this case, the site `z` can have one of two different states `p` and `q`.
A rule provides a precise description of how agents interact. Such a rule is applied, when

a match to its precondition is found. A precondition within the context of Kappa is a certain combination of agents, sites and links between sites. Rule application means that the model entities, represented by a match, are modified in such a way, that they satisfy the rule's postcondition. Listing 2.2 contains some basic example rules using Kappa syntax. In the Kappa language links between agents, described in rule preconditions, are expressed through link states inside brackets, next to sites. Site states are written in curly brackets. The arrows in listing 2.2, separating pre- and postcondition, indicate the directionality of a rule. A rule can either be unidirectional `->` or bidirectional `<->`. The latter essentially means that a rule can be applied backwards, with its postcondition as precondition and its precondition as postcondition. The `@` operator, accompanied by a number at the end of a rule, denotes the rate constant k. It is the probability that the rule is applied in the time interval between t and $t + dt$, given that its precondition is met. Furthermore, a rule can either be atomic, i.e., it performs only one type of action, or it can be non-atomic and combine several actions. Danos et al. [DFF+09] defined 5 five classes of atomic rules, which may perform one of the following actions: binding, unbinding, modification, creation and deletion.

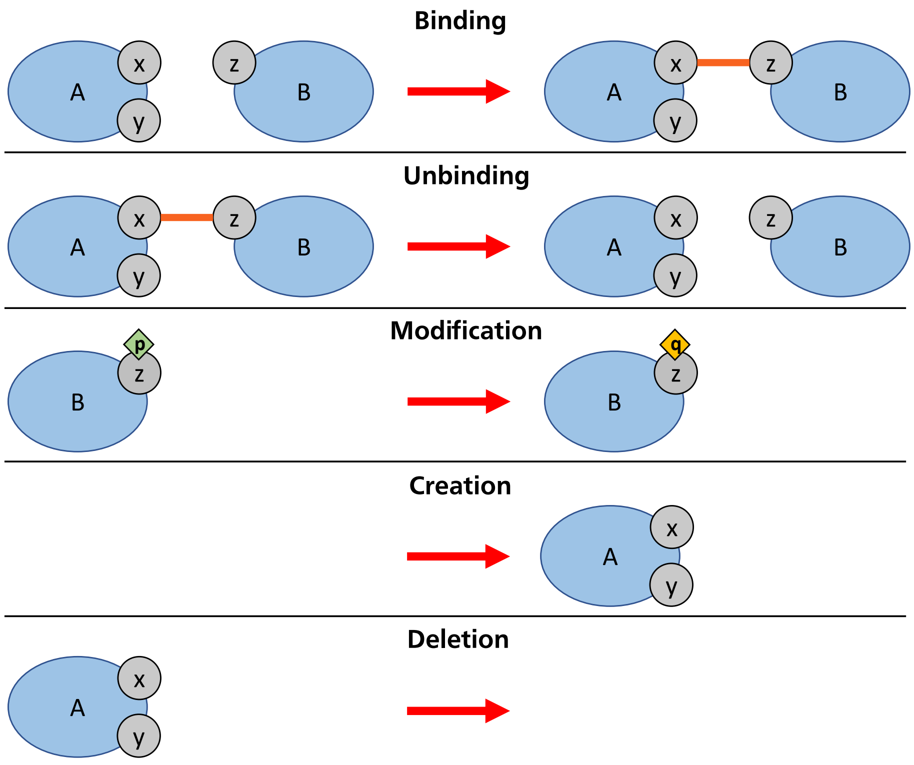


Figure 2.3: Kappa – Atomic Rules

The first class of atomic rules binds two agents, as illustrated in figure 2.3 and `rule_1` in listing 2.2 line 1. In this example, `rule_1` defines two agents which are not yet bound, i.e., unbound, as its precondition. Unbound sites of agents are denoted by the dot in brackets. When the rule is applied, it connects agents `A` and `B` at site `x` and `z` through a link, according to the link state with index `1`.
The unbinding of two agents is the second class of atomic rules and illustrated in figure

2.3 and listing 2.2 line 2. For example, `rule_2` defines two bound agents, denoted by the link with index 1, as its precondition. When the rule is applied, it deletes the link and therefore disconnects agents `A` and `B`, at site `x` and `z`.
The third class of atomic rules is the modification of site states. When looking at the precondition of `rule_3`, in listing 2.2 line 3, we notice that site `z` of agent `B` is in state `p`. As opposed to the postcondition of `rule_3`, where site `z` is in state `q`. Consequently, when the rule is applied the site state is changed from `p` to `q`, as illustrated in figure 2.3. As we can see in figure 2.3, atomic rule class 4 creates and class 5 deletes an agent. In Kappa, the creation of an agent is expressed by writing a dot, instead of an agent as a precondition and the to be created agent is defined in the postcondition. Deletion works the other way round. Instead of creating a new agent through rule application, an agent is deleted. The to be deleted agent is described in the precondition, while the postcondition receives a dot (see listing 2.2 lines 4 and 5).

Listing 2.2: Atomic Rules

```
1  'rule_1' A(x[.], B(z[.]) -> A(x[1], B(z[1])       @ 1      //Binding
2  'rule_2' A(x[1], B(z[1]) -> A(x[.], B(z[.])       @ 1      //Unbinding
3  'rule_3' B(z{p}[.])      -> B(z{q}[.])            @ 1      //Modification
4  'rule_4' .               -> A(x[.], y[.])         @ 1      //Creation
5  'rule_5' A(x[.], y[.])   -> .                     @ 1      //Deletion
```

As a remark, Kappa defines three additional link states, besides a dot and an index. Listing 2.3 shows some exemplary rules, with patterns that make use of these link state types. In the first example, `rule_3_1` makes use of the wild card, denoted by `#`, which implies that the link state of site `p` is of no relevance to this precondition. Example `rule_3_2` demonstrates the *bound-to-any* link state, denoted by `_`, stating site `p` must be bound another site, which one does not matter. Lastly, example `rule_3_3` uses the *bound-to-any-of-type* link state which refines the previously shown link state and is expressed through `<Site>.<Agent>`. This link state requires site `p` to be bound to another site of type `x`, belonging to an agent of the specified type `A`.

Listing 2.3: Additional Site States

```
'rule_3_1' B(z{p}[#])   -> B(z{q}[#])   @ 1      //Wild card
'rule_3_2' B(z{p}[_])   -> B(z{q}[_])   @ 1      //Bound-To-Any
'rule_3_3' B(z{p}[x.A]) -> B(z{q}[x.A])@ 1      //Bound-To-Any-Of-Type
```

For any simulation to be able to run, initial conditions are required. In Kappa, initial conditions are defined through the `init` keyword, followed by the agents that should be instantiated and the number of instances. Listing 2.4 line 1 shows an example instantiation of 10 instances of agent type `A`.

Listing 2.4: Initial Condition and Observable

```
1  %init: 10 A(x[.], y[.])
2  %obs: 'num_of_B' |B(z{q}[#])|
```

The last feature presented in this subsection are the *observables*, expressed through the `obs` keyword of the Kappa language. Observables are used to get simulation statistics, such as the number of current instances of a certain agent pattern or type. An observable is defined using the `obs` keyword, followed by a label name and the agent pattern that should be observed, i.e., printed out. For example, the observable in listing 2.4 line 2 leads to a printout of the total amount of agent `B` instances, with their sites `z` being in state `q`.

As a final remark, Kappa has many more features, besides the few named in this subsection, but these are not essential to this thesis. Most of the other not mentioned features are for convenience purposes, such as variables, or are extensions to Kappa's expressiveness, like arithmetic expressions.

2.3 Model-Driven Software Engineering

Since this thesis aims to simulate biochemical processes in a rule-based fashion, rules, agents, bonds between agents and the simulation environment itself have to be modeled. One could go about this in a straight forward approach, for example, modeling the reaction process in an algorithmic fashion and apply it to a population of agents. However, this approach will be very tedious to maintain, when we start to modify either rules or agent types, and would have to be partially redone, if we want to simulate a different process. The approach of **M**odel-**B**ased **S**oftware **E**ngineering (*MBSE*) naturally lends itself to tackle the problem of re-usability and reconfigurability. MDSE can be defined as a set of instruments and guidelines for applying the advantages of modeling to software engineering activities [BCW12]. The core concepts of MDSE are models and transformations, with the latter meaning to apply manipulation operations of any kind to a model. MDSE follows the statement "Everything is a model" very closely, which implies that we do not only have models representing data, but we also think of model transformations as models themselves. To create models within the context of MDSE a modeling language is employed (e.g., UML), which in turn can be defined by a model itself. In MDSE this is called meta-modeling, i.e., modeling a model, which can also be interpreted as modeling all possible models that can be expressed with a modeling language.

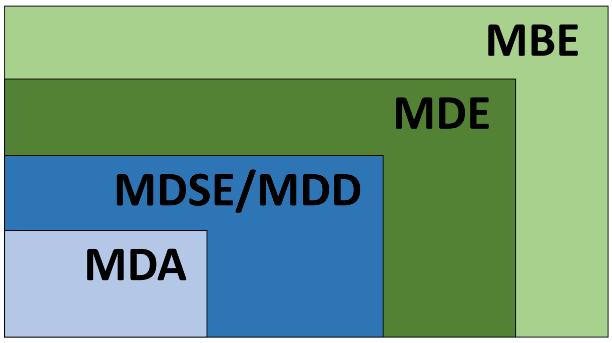


Figure 2.4: Taxonomy of MD* Paradigms

There is a plethora of terms in the model-driven universe, that for the most part mean the same or represent an umbrella term for different aspects of the model-driven paradigm. The most commonly used term is **M**odel-**D**riven **E**ngineering (*MDE*), which encompasses more than pure software development activities and includes other model-based tasks of a software engineering process, e.g., model-based evolution of a system [BCW12]. As shown in figure 2.4, MDE represents a superset to MBSE and **M**odel-**D**riven **S**oftware **D**evelopment (*MDSD*), also called **M**odel-**D**riven **D**evelopment (*MDD*). The latter focuses on using models as primary tool of the development process and generating the implementation automatically from models. **M**odel-**D**riven **A**rchitecture (*MDA*) is a specialization of MDSE/MDD, created by the **O**bject **M**anagement **G**roup (*OMG*) and provides a standard for modeling and transformation languages. The OMG standard promotes the use of modeling architectures, like the **M**eta-**O**bject **F**acility (*MOF*) [Obj05]. **M**odel-**B**ased **E**ngineering (*MBE*) lies on the other side of the spectrum, in which software models are important, but not the driving factor in development. Hence, MBE is a superset to MDE.

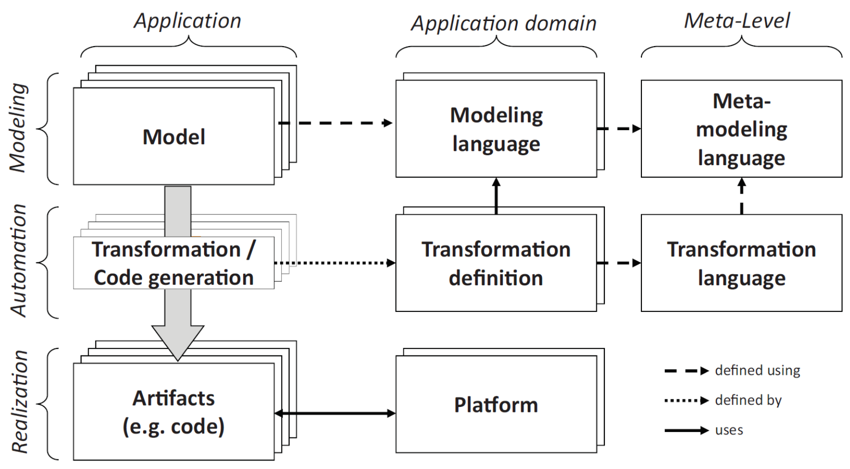


Figure 2.5: MDSE Methodology [BCW12]

The main aspects of MDSE are illustrated in figure 2.5, where MDSE approaches problems in orthogonal dimensions. In the figure conceptualization is represented by columns and implementation by rows. Implementation tackles the issue of mapping models to systems and consists of three core aspects:

- On the *modeling level* models are defined.
- Mappings from modeling to realization levels happen on the *automation level.*
- Implementation of solutions through artifacts takes place on the *realization level.*

Conceptualization on the other hand defines conceptual models for describing reality and consists of three core aspects as well:

- Definition of application models, model transformation and generation of running components happen on the *application level.*
- On the *application domain level* the modeling language is defined as well as transformations and implementation platforms for a specific domain.
- Conceptualization of models and transformations takes place on the *meta-level.*

As shown in figure 2.5, the MDSE work flow starts with a model and through the use of model transformations finishes with a realization of the given model, e.g., running code in the case of software development. Models are specified according to a modeling language, which itself is defined by a meta-modeling language. Transformations are described through a set of transformation rules, in turn defined by a transformation language. Furthermore, the above described various layers of abstraction make models and transformations platform independent. That means, models are often described through some kind of platform independent markup language, e.g., XML. Platform dependent artifacts are only generated in the final step, i.e., the realization (see figure 2.5). This allows for models to be reused and systems to run on different platforms.
As one can see, this makes for a very compelling concept for creating reusable, interchangeable software, which prevents many compatibility issues. Additionally, modeling grants several perks, for example, syntactical validation, model checking and model simulation [BCW12].

2.3.1 Modeling and Metamodeling

A model is usually defined by using a modeling language. Such a tool allows the designer to define a representation of a conceptual model, either through graphical representations, textual specifications, or both.
We can defines two classes of languages [BCW12]: **G**eneral-**P**urpose **M**odeling Languages (*GPL*) and **D**omain **S**pecific **L**anguages (*DSL*). Languages of the former kind are modeling tools that can be applied to any problem domain for modeling purposes. The UML language used to describe, e.g., software architectures, is a well known example as well as state machines, control flow diagrams and many more.
DSLs, on the other hand, are languages that are tailored specifically to suit a certain domain and are used to describe entities in that specific domain. Latex is an example of a DSL, which is used for type setting and formatting text. Database languages like SQL, hardware description languages like Verilog, or MatLab used in mathematics, represent another three of many examples for widely used DSLs.

As stated in 2.3, models are an integral part of the MDSE paradigm. When we look at models from the same domain, we notice that they share common traits. For this reason, it is only natural to see models as instances of more abstract models. These are called metamodels and represent another layer of abstraction, the same way that models represent an abstraction of reality. If we continue along this train of thought, even metamodels themselves can be seen as instances of meta-metamodels. For example, a model of a tool used to create a DSL, e.g., Xtext, is a meta-metamodel to metamodels defined by the DSL. This process of metamodeling can be repeated recursively for infinite levels. In practice, it has been shown that meta-metamodels can be defined based on themselves [BCW12], which makes it impractical to go beyond this level of abstraction.
The principle of metamodeling is illustrated by figure 2.6, which shows parts of the metamodel used in the implementation section of this thesis and contains entities like agents that have been introduced in previous sections. The presented 4-layer architecture [BCP12] is typical for MOF based projects.

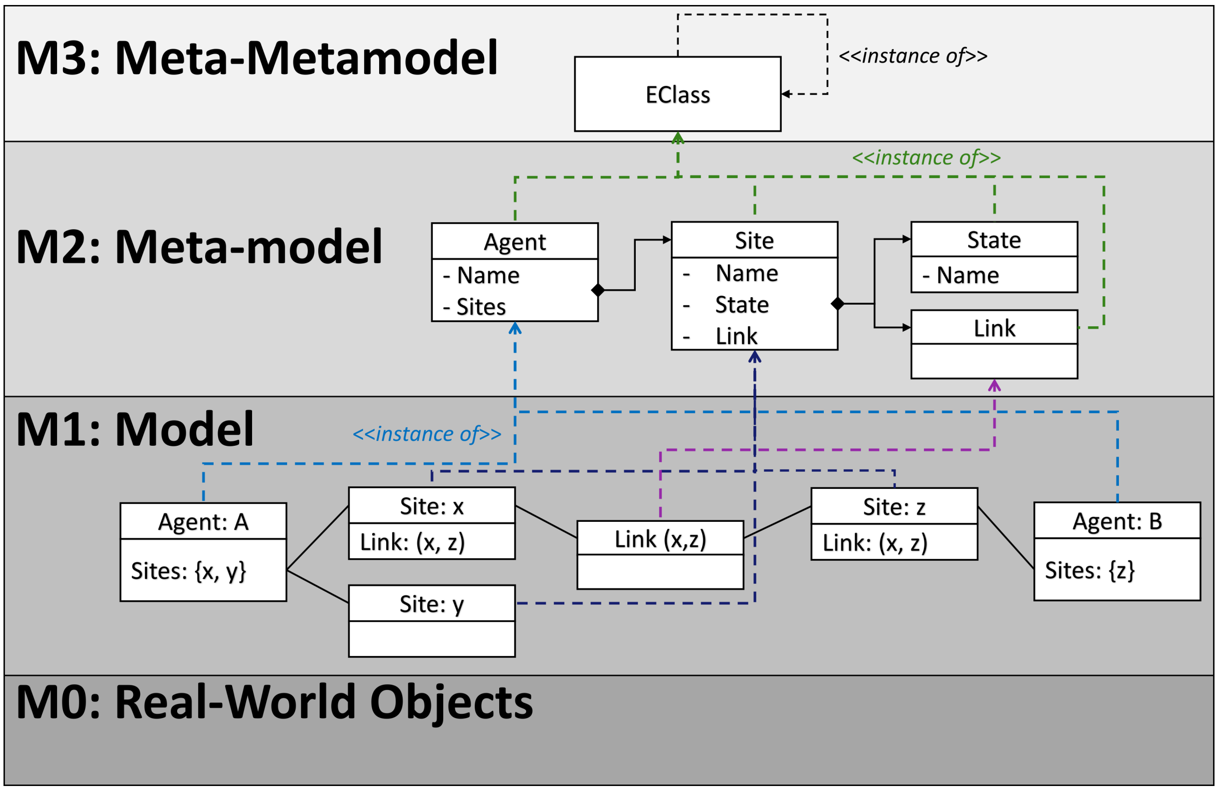


Figure 2.6: Metamodeling

- **M3**: The *Meta-Metamodel Layer* contains the modeling language used to describe the structure of the metadata. In this case, a MOF implementation called **E**clipse **M**odeling **F**ramework (*EMF*[4]) provides the meta-metamodel at M3. EClass represents the most important entity on this level, all entities in M2 are instances of this base type.

- **M2**: The *Metamodel Layer* provides definitions for the in M3 described structures of meta-data. M2 is where a modeling language, e.g., UML, or a DSL is defined. In figure 2.6, the metamodel layer specifies `Agent`, `Site`, `State` and `Link` entities, which may be used to express relations between `Agent` instances, through linked `Site` instances.

- **M1**: The *Model Layer* contains definitions of data and is the layer where modeling languages, or DSLs operate. They define the properties of models, i.e., the structure of elements in the M1 layer. In the example, `A` is an instance of an `Agent` entity from M2 and it contains two instances of the `Site` entity. Agents of type `A` and `B` are linked, because their contained sites share the same `Agent` entity. Therefore, **Link** entities are used to connect agents at their sites.

- **M0**: The *Information Layer* contains objects, i.e., instances of M1 entities. They serve as template for real world objects. Hence, M0 represents the actual data that we wish to describe.

4 EMF project page: https://www.eclipse.org/modeling/emf/

2.3.2 Model Transformations

The previous sections looked at models in a static fashion, where model entities represented some kind of object or property of the real world. In other words, the models were static and could not change. Regarding the goal of this thesis, i.e., creating a rule-based simulation, we need the models to be able to change and entities within models should be able to interact with each other. For this reason, we need model transformations.

As stated in subsection 2.1, besides models, transformations are equally important and represent another crucial ingredient to MDSE. The term transformation describes more than one type of operation that can be performed upon a model. Typical operations are: **M**odel-to-**M**odel (*M2M*), **M**odel-to-**T**ext (*M2T*) or **T**ext-to-**M**odel (*T2M*) operations. M2T transformations receive a model as input and generate a text string as output. T2M transformations, on the other hand, take text as input and generate models as output.

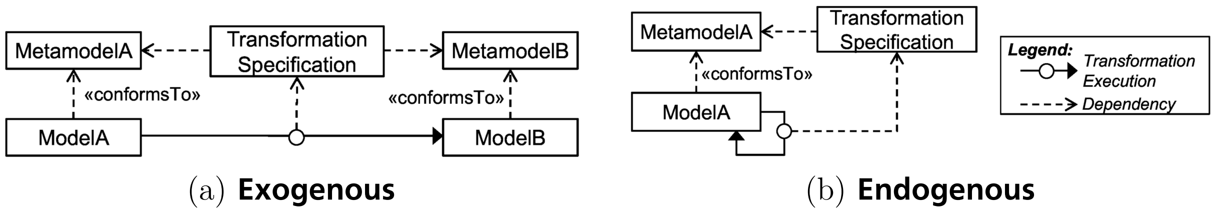


Figure 2.7: Types of Model Transformations [BCW12]

M2M operations can be used to migrate models between different metamodels or to refine models [ELS+10]. These operations are also called *exogenous* or *endogenous* model transformations [MG06], respectively. The former concept is illustrated in figure 2.7(a), where `ModelA` from `MetamodelA` is transformed to `ModelB` conforming to the specifications of `MetamodelB`. A possible scenario, where such a transformation would be useful, is the translation between two different programming languages. For example, `MetamodelA` could be Java and `MetamodelB` represents C++. The goal would be to transform program A, written in Java, to a program B, with the same semantics, but written in C++. Figure 2.7(b), on the other hand, shows the concept of model refinement, where a `ModelA` from `MetamodelA` is transformed to a model that still conforms to `MetamodelA`. Looking at the metamodel example in figure 2.6, we can think of the following possible scenarios for refinement: Two or more `Agent` instances have to be linked, created, destroyed or unlinked, similar to atomic rules in Kappa, shown in subsection 2.2.2. In essence, such transformations are required whenever changes to a model need to be expressed.

A basic M2M application is a *source-to-target*, or *target-to-source* transformation. In this case, an existing model is used as input, is then modified and finally a new output model is generated. An incremental model transformation represents a more advanced M2M use case, which improves upon the former method and only updates the source or target model, without completely regenerating them. The last scenario is model synchronization, where source and target models can be modified, and consistency is kept by propagating changes from model to model. For this reason, keeping models consistent is an important task but as it turns out, a non-trivial one. For this reason, it is still subject of ongoing research. As a remark, **T**riple **G**raph **G**rammars (*TGG*) [Sch95], which are already in frequent use, solve this problem by defining a correspondence graph between two metamodels. This correspondence model can be used to synchronize models and check for consistency as well as allowing model transformations in both directions.

Typical applications in MDSE scenarios frequently employ exogenous M2M transformations, where models are often manually modified through modeling editors or DSLs, by editing, creating and or removing entities in the model. Regarding the goal of this thesis, i.e., rule-based simulation, an exogenous approach does not make sense, because we do not want to translate between models that conform to different metamodels. Instead, we want to refine models conforming to some metamodel iteratively, through transformations. Additionally, these transformations should be performed in-place, because building the output model from scratch after every transformation step would be impractical, since there might be many steps. A transformation is in-place, when changes to a model are applied without copying all the static parts of the model, i.e., the parts that are not affected by the transformation. For this reason, endogenous in-place transformations are most suited for simulation purposes in an MDSE context.

Graph transformation [EEPT06] is a common approach to in-place transformations. It is a declarative, rule-based technique for expressing in-place model transformations, based on the fact that models and meta-models can be expressed as graphs. Therefore, they can be manipulated using graph transformation techniques.
Graph transformations have several perks that make them so useful. Their general nature allows for in-place transformations as well as the formulation of out-place transformations, by incorporating source and target models into one graph. Furthermore, graph transformations have a visual form, which makes them and the rules they are representing, intuitive. Finally, their formal nature makes it possible to subject rules to analysis. A set of graph transformation rules and an initial graph, on which a transformation according to the rules is performed, is called a graph grammar. Graph transformation rules, henceforth, called rules, contain a **L**eft-**H**and **S**ide (*LHS*) and a **R**ight-**H**and **S**ide (*RHS*). The left-hand side graph represents preconditions which must be met, such that the rule can be applied (the rule fires). The right-hand side graph, also called postcondition, represents the outcome of the rule application. Equal identifiers on both sides mark corresponding elements on LHS and RHS. The application of a rule, whose preconditions have been met, performs the following modifications [BCW12]:

- All elements, present only on the LHS, are deleted
- Elements that exclusively reside on the RHS are added
- Elements that exist on the RHS and the LHS are preserved

Using just the LHS to specify preconditions only enables us to describe the existence of certain nodes in a subgraph. That means, we can not prevent the presence of unwanted nodes. For this reason, **N**egative **A**pplication **C**onditions (*NAC*) are introduced, which

specify the elements that must be absent from a subgraph, in order to fulfill a precondition of rule. In short, the LHS specifies elements that must exist in a matching subgraph, while the NACs describe what must not be attached to a matching subgraph, such that a rule may be applied to it.
As stated before, in order for a rule to be applicable the preconditions have to be met. That means, a subgraph matching the graph specified in the LHS and conforming to all NACs has to be found in the initial graph of the graph grammar. The retrieved subgraph is called a match, while the retrieval process itself is called pattern matching, which is explained in further detail in subsection 2.3.3. The rule is then applied to the found match. If there is more than one match, a random one is chosen. During the application, contents of the match are modified to fulfill the postcondition, specified by the RHS of the rule body. Graph transformation rules of the grammar are usually applied as long as possible, i.e., until no further matches for any rule precondition can be found. Furthermore, rule application may occur in any order (non-deterministic), but can be influenced by assigning priorities to certain rules.

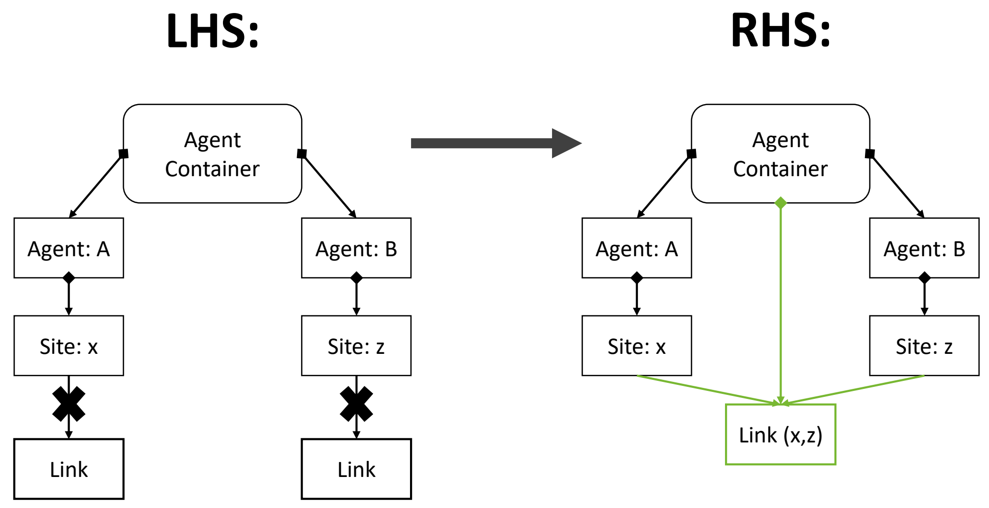


Figure 2.8: Graph Transformation

Figure 2.8 illustrates a graph transformation, using a model that conforms to the meta model shown in figure 2.6. In the example, the rule's LHS is a connected graph consisting of two `Agent` instances of type `A` and `B`, each having a site. Sites `x` and `z` may not have a reference to a `Link` object, i.e., they must not be connected. This is depicted by the crossed out reference arrows in figure 2.8 and is an example of a NAC. Consequently, matches are discarded, if sites `x` and `z` are already connected to something else. When a match to this LHS is found, the rule is applied to the subgraph of the model. In this case, a `Link` object owned by an `Agent Container` object is created, with sites `x` and `z` referencing the link and by virtue of that, connecting agent `A` and `B` through sites `x` and `z`. Therefore, the rule application has modified the LHS subgraph in such a way that it satisfies the rule's RHS.

In this thesis, only model-to-model transformations will be relevant. Therefore, the term transformation will, henceforth, imply this type of operation. Furthermore, model transformations used herein refine models, they do not translate between different metamodels. As a final remark, transformations within the context of this thesis will be performed in place.

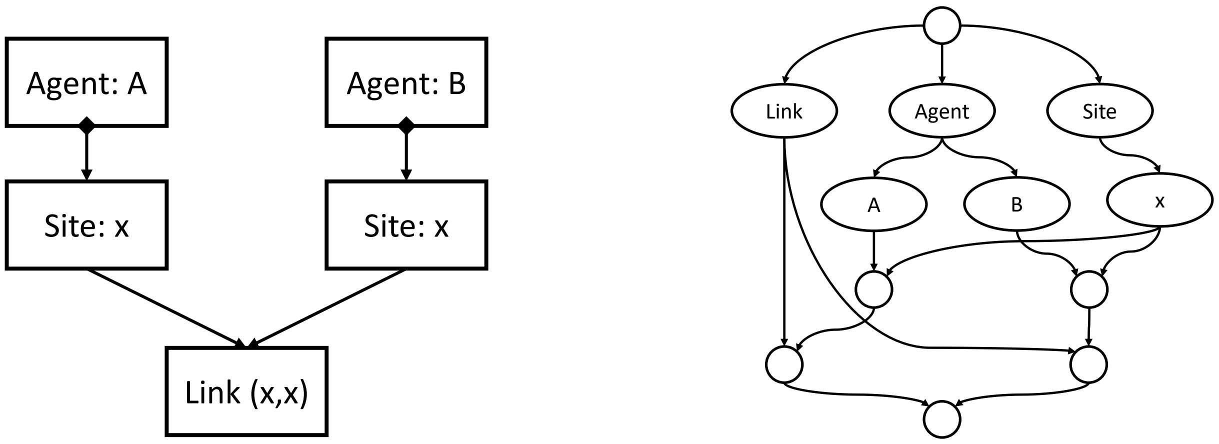
2.3.3 Pattern Matching

The term pattern matching is widely used and appears in numerous different problem categories of computer science, which are often only slightly related. For example, **C**omputer **V**ision (CV) is a popular domain where the term pattern matching frequently appears in. Patterns in CV often represent an image patch or some kind of feature descriptor, both represented by some vector. Hence, pattern matching in the aforementioned case refers to the process of finding the best match to the feature descriptor in an image matrix, according to some metric.
Another frequent use case for pattern matching is graph transformation, which plays an important role in this thesis. As described in the previous subsection, graph transformation rules have a precondition and a postcondition. The LHS, i.e., the precondition of a rule, defines a subgraph that must appear in the initial graph, i.e., the model, in order for a rule to be applicable. Therefore, finding these subgraphs in a model is an important aspect in model transformation. Since the previously mentioned subgraphs represent reoccurring patterns in the initial model, the process of finding those is called pattern matching as well.
As a remark, the term pattern matching only refers to the recognition of reoccurring patterns in a dataset. It does not state how patterns are defined, how they look or even how they are retrieved. Henceforth, this thesis will use the term pattern matching exclusively within the context of graph transformations.
There are quite a few approaches to pattern matching on graphs, such as batch and incremental pattern matching approaches, which will be explained in the following paragraph.

Batch pattern matcher pursue a rather straight forward principle. First, the set of all matches to the LHS sub-graph of a rule is found in the initial graph. From this set, a random match is selected. These matches are usually found with pattern matching algorithms based on solving a constraint satisfactory problem [LV02] or through local search algorithms using search plans [Zü96]. The next step is checking for potential NACs, which might void the previously found match. Eventually, a graph transformation engine modifies the match according to the rule's RHS, i.e., it replaces the found sub-graph inside the model with a copy of the rules RHS sub-graph. For each subsequent graph transformation rule, all previously found matches are discarded and the above described process is restarted from scratch.
Experiments in benchmarking graph transformations [VSV05] have demonstrated that batch pattern matching negatively influences performance, in certain situations. As models and potentially LHS patterns grow in size, this traditional approach to pattern matching can become a severe performance bottle neck. Such performance problems can appear in application scenarios, where models containing large amounts of entities undergo rapid changes. This forces the pattern matcher each time to discard and regather matches, by searching through a large model.

Incremental pattern matcher on the other hand try to mitigate the negative performance effects of batch pattern matching on graph transformations, by not discarding all found matches after every rule application. Instead, changes to the model are registered and the set of matches is updated incrementally, according to those changes. Initially, most approaches to incremental pattern matching only provided partial support for NACs. Later, a working concept was presented by Varrò et al. [VVS06], in which their incremental pattern matching approach was described.
In the initial step, i.e., the preprocessing phase, all matches to LHS patterns as well as partial matches are collected and stored in a matching tree. For this a search plan is used, which knows how to store and construct matches from partial matches, by traversing the matching tree. The *Rete-Network* is an example of such a matching tree. After that, each modification that is applied to the model, triggers an incremental update on the matching tree. Subgraphs that match NACs are stored and used to invalidate corresponding LHS matches, during each update. An in-depth explanation to the procedure is presented in a different paper by Varrò et al. [VVS].

Rete-Networks were designed to make many pattern matching less expensive and were originally developed for use in system interpreters [For82]. The goal of the *Rete Match Algorithm* is to find matches to a set of LHS patterns in an existing model, without having to iterate over all elements of model, each time it changes.



(a) **Example Pattern** (b) **Example Rete-Network**

Figure 2.9: Rete-Networks (1)

The Rete Match Algorithm achieves that goal by generating a decision-network in the shape of a data flow graph from the given set of LHS patterns. Nodes in this graph have two basic types: *selection* and *junction*. Elements from the model on which the pattern matcher should search for matches, i.e., the working memory, are passed through the network. Selection nodes remember and pass on elements that satisfy certain criteria, e.g., the possession of certain attributes or types. Junction nodes, on the other hand, take the output from selection nodes and merge them. For example, in the pattern “A Person

with name X and age Y” a selection node would check the name, i.e., variable X, and another one the age, i.e., variable Y. A junction node would in turn represent the “and”, by joining the output of both selection nodes. The output of such a network would be a match to the example pattern, containing any person object that has a certain name and age, depending on the value of the variables X and Y.
For a better understanding of the inner workings of a Rete-Network, consider the following example: Figure 2.9(a) shows an exemplary pattern conforming to the meta-model in figure 2.6, from subsection 2.3.1. It is a basic pattern, which describes two agents of different types, each possessing sites of a certain type that are connected through a link. In figure 2.9(b), a data flow graph is created from the given pattern. As we can see, there are selection nodes for the object type, followed by selection nodes for agent and site type attributes, respectively. Junction nodes join the output in such a manner that agents have their correct type, sites and site types. The lowest junction node joins sites that posses the same link object, i.e., they are linked, and represents the output of the network. Every element of the model that is saved in the final node is part of a match to the given pattern.

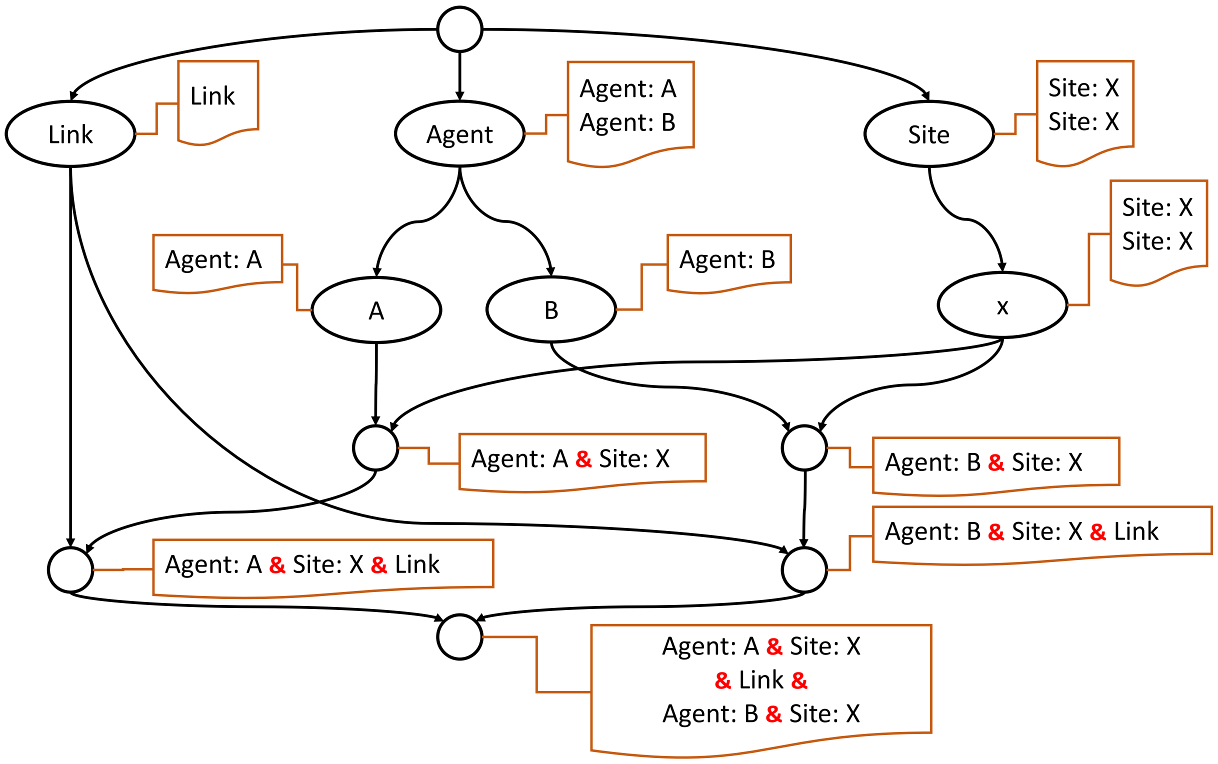


Figure 2.10: Rete-Networks (2)

The Rete Match Algorithm does not search the working memory for new elements to pass through the network. Instead, as new elements are added to the model, they are passed to the Rete-Network for indexation. This process is illustrated by figure 2.10, where a model subgraph conforming to the pattern from the previous example is passed through the network. Each node possesses its own table. Elements entering the node are stored when they match the node’s criterion and passed on to its successor nodes.

Elements that leave the working memory are passed to the Rete-Network as well. However, passing through, the network does not add matches to internal node tables, but removes them.

As we can see, incrementally updating a graph structure is much more efficient than discarding every information gathered on a model, whenever it has changed. On the down side, Rete-Networks grow rapidly with increasing numbers of patterns and pattern sizes. Additionally, each node in a Rete-Network carries its own table for partial matches. Considering these facts, in certain situations Rete-Networks may consume large amounts of memory, but in turn can offer much better performance, with respect to speed, compared to classic pattern matching approaches. Rule-based simulation is one of those situations, since it implies a large model undergoing frequent transformations, which would force the batch approach to discard and regather information in rapid succession. A batch approach might be faster in the first iteration of such simulations, but would eventually trail behind the incremental approach after subsequent iterations. Therefore, this thesis relies on tools that perform incremental pattern matching on instance graphs.

3 Implementation

This subsection presents the implementation of a new framework for rule-based stochastic simulations of biochemical processes. The resulting software is composed of several modular components. Therefore, the following subsection shows a brief overview of the architecture, the interfaces, the involved components and explains the reasoning for the high degree of modularity. After the overview, a new domain-specific language is presented, which was developed to model reactions as rules, agents and initial conditions of the simulation. The third subsection of this chapter is concerned with pattern matching. It details which general-purpose pattern matching tools are used, how they are embedded in this framework and which role they are playing in the context of rule-based simulation. The last subsection explains how the simulation itself is working, by detailing the necessary steps in each iteration.

3.1 Overview

As depicted in figure 3.1, the simulation framework can be divided into three major aspects: *Meta-Modeling*, *Configuration* and *Simulation*.

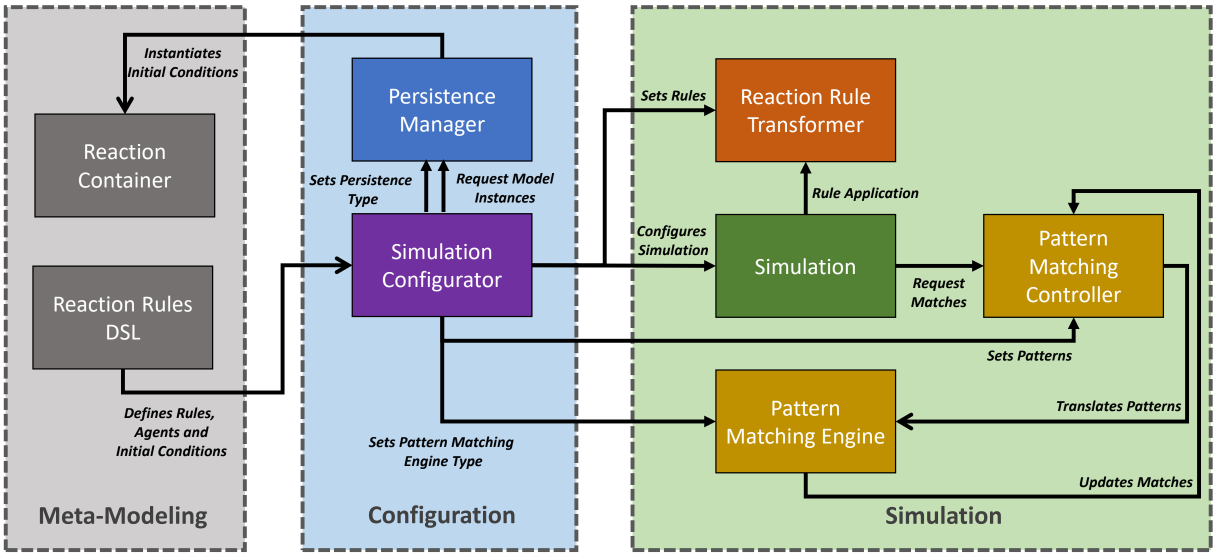


Figure 3.1: Simulation Framework Overview

The aspect of meta-modeling encompasses the *Reaction Container* metamodel and the *Reaction Rules* domain-specific language. The DSL is used to specify agents and rules, which are applied to agent instances, contained in Reaction Container instances. Details concerning the meta-modeling part of the implementation are further discussed in subsection 3.2. The simulation aspect of the framework, which is explained in detail in subsection 3.3, covers all modules and functionality required during runtime, i.e., the actual core of the rule-based simulation. This includes pattern matching, model transformation through rule application, checking for termination conditions and recording simulation statistics. The final aspect, configuration, implements the idea of a modular framework. It comprises model persistence, i.e., saving and loading of models, and the ability for the user to activate, deactivate or change modules of the simulation framework.

Modularity plays a big role because, besides the goal to enable a rule-based simulation, we would like to evaluate how each general-purpose pattern matching tool performs, while using patterns of the biochemistry problem domain. For this reason, the *Simulation* module is designed in such a way that it is agnostic to the used pattern matcher and can, therefore, be exchanged.
Another reason for modularity is the fact that some patterns, as they appear in this problem domain, have turned out to be problematic for general-purpose pattern matching tools. Hence, the **P**attern **M**atching **C**ontroller module (*PMC*) is introduced, which serves as an abstraction layer between the patterns, defined in the DSL, and the actual patterns given to the pattern matcher. This makes it possible to apply preprocessing steps to patterns in order to improve pattern matching performance. Furthermore, to enable the investigation of possible performance gains, the PMC is designed to be interchangeable. The aforementioned problematic patterns, the causes for and solutions to bad performance of the used pattern matcher, are further discussed in subsection 3.4.

The interaction between modules as well as their functions, can best be explained by looking at a typical configuration work flow, which produces a simulation of a biochemical process as a result. For a better understanding, the mentioned modules and their interactions are shown in figure 3.1.
Initially, the model describing agents, rules and initial conditions of the simulation is created using the DSL. With the help of the *Simulation Configurator* module, using the specified model name, this model can be loaded by the *Persistence Manager* module. Furthermore, the Persistence Manager module optionally works with a graph database or conventional XML files. The preferred persistence method can be chosen with the help of the Simulation Configurator. Then, a Reaction Container instance can be loaded, which is generated on-demand by the persistence module and in which the actual simulation takes place. The Reaction Container has a set of agent instances, in which the number and configuration of the instanced agents correspond to the previously defined initial conditions. Furthermore, the Simulation Configurator determines which general-purpose pattern matching tool is used in the background. Additionally, it can be chosen if the patterns, appearing in the transformation rules, should be preprocessed to improve performance.
Once all these configurations have been carried out, any number of simulation instances can be created. Settings are automatically applied to simulations by the Simulation Configurator upon creation.

3.2 Reaction Rules DSL

Before we are able to perform any kind of simulation, we must first determine the properties of the entities within our simulation, describe how they can interact and define how many entities exist, in which state, at the beginning. In case of a biochemical simulation, the simulation participants are biomolecules, more precisely proteins. Since this thesis uses the rule-based modeling approach, explained in subsection 2.2, proteins are abstracted by so-called agents. Hence, reactions between molecules are represented by interactions between agents. These interactions are modeled through rules, whose preconditions correspond to certain agent configurations. If such a configuration of agents is found in the model of the running simulation, the rule is triggered and the found agents are modified in such a way, that they correspond to the configuration of the postcondition (see subsection 2.3.2). We can see that the step of modeling plays an important role and may turn out to be quite complex, regarding the problem domain of biochemistry. For this reason, a new DSL is presented in this thesis, which is used to create simulation models.

This new domain-specific language, henceforth, called *Reaction Rules DSL*, offers the means to define the above described simulation aspects, like agents, initial conditions and rules. In addition, termination conditions which end the simulation once they are met, and agent configurations, which should be observed during simulation, can also be defined through the Reaction Rules DSL.

The DSL was created with the help of Xtext[5], which is a plug-in for the Eclipse Java IDE[6] and is a popular tool for creating custom DSLs. In Xtext, the grammar and the syntax, e.g., the keywords of a language, can be defined textually using a context-free grammar. Given this, Xtext creates a model, which in turn is a metamodel to which all models created with the DSL must conform to. The advantage of this approach is that all modules, working with models that were created using this language, can rely on the fact that they conform to the same language metamodel. For example, looking at the DSL of this thesis, agents can only be connected to each other through sites, because these models must conform to their metamodel. The here presented DSL has a total of six keywords: `agent`, `rule`, `init`, `obs`, `terminate` and `var`, which are explained in the following paragraph.

Listing 3.1: Agent Declaration

```
1  agent A(x,y)                          // Declaration of agent A
2  agent B(z{p, q})                      // Declaration of agent B
```

Analogous to Kappa, presented in subsection 2.2.2, an agent is defined with the keyword `agent`, which determines the characteristics a possible agent instance can have. For example, listing 3.1 in line 1 shows the definition of an agent, with the name `A` and two sites `x` and `y`. Similar to Kappa, sites enable connections between agents, by sharing the same link. Site states, on the other hand, serve the purpose of representing different conditions of certain sites. For example, listing 3.1 line 2 shows how an agent `B`, with a site `z` and two possible states `p` and `q` is defined. It is also possible to define more than two states or no state at all, as is the case with agent `A` in the first example. The DSL requires agent identifiers, i.e, names, to be unique. However, the site identifiers must only be locally unique, i.e., within the context of their agent. The same applies to states, which only need to have a unique identifier within the context of their site.

Another important component are rules, as mentioned at the beginning. In order to

[5] Xtext project page: https://www.eclipse.org/Xtext/

[6] Eclipse project page: https://www.eclipse.org/ide/

define a rule, the keyword `rule` must be used. As shown in listing 3.2, an identifier must first be defined for the rule, followed by the LHS, the direction operator, the RHS and the application probability. LHS and RHS contain the agent configurations described earlier. These agent configurations are, henceforth, referred to as *agent patterns*. Agent patterns describe the set of agents a subgraph must have in order to match the pattern. Each agent may also contain a set of sites, with each site potentially having a certain state. The agent types, defined through the use of agent keyword, can be referenced in such patterns and describe links between agents via sites. Such links are defined using link states, which describe links between two sites in a pattern, identified through an index that is unique within such a pattern. The uniqueness of the index within a pattern is important in order to later correctly identify linked sites within a pattern, using the index. The link state is written inside brackets, on the right side of a site, whereas the site state is written in curly braces, on the left. Only sites and states described in the agent definitions can be used for this purpose, which is intended to prevent semantic errors in the simulation. The directional operator of a rule indicates, whether it is a bidirectional `<->` or unidirectional `->` rule, which means that the rule can be applied forwards and backwards, or only forwards. A bidirectional rule basically works like an unidirectional rule, except that the RHS and the LHS trade places, in case of a backwards rule application. The last parameter of a rule, the application probability, specifies the probability that a rule is applied, if an instance of the LHS is found in the model.

Listing 3.2: Basic Rules

```
1  rule r1 {A(x[free], B(z[free])} -> {A(x[1], B(z[1])} @ [1]
2  rule r2 {A(x[1], B(z[1])}        -> {A(x[free], B(z[free])} @ [1]
3  rule r3 {B(z{p}[free])}          -> {B(z{q}[free])} @ [1]
4  rule r4 {void}                   -> {A(x[free], y[free])} @ [1]
5  rule r5 {A(x[free], y[free])}    -> {void} @ [1]
```

Analogous to Kappa (see subsection 2.2.2), rules can be divided into 5 basic actions: binding, unbinding, modification, creation and deletion. All basic actions can also arbitrarily be combined in one single rule.

For example, listing 3.2 line 1 shows the binding of two agents. An agent `A` is connected at site `x` to another agent `B` at site x. This link is uniquely identified by index 1. The precondition of this rule states that the sites `x` of the `A` type and `B` type agent must not be connected. This is indicated by the keyword `free`, which improves readability, in contrast to Kappa, where a simple dot is used. In addition, agent patterns are always placed in curly brackets, which is also a measure to improve readability.

The example in line 2 shows an unbinding, in which the action from line 1 is undone. As a remark, basically, both rules `r1` and `r2` could be expressed through a single rule, by replacing the unidirectional operator with the bidirectional operator in line 1. In this case, a second rule application probability would have to be added as a parameter in the square brackets, with the same probability as `r2` in line 2.

An example of a modification is shown in line 3, where the state of site `z` of an agent `B` is set from `p` to `q`.

A creation rule implies the generation of one or more new agent instances, as defined by the RHS. If the rule is then applied to a subgraph matching the LHS, new agent instances

are added to the model. In Listing 3.2 line 4, an exemplary creation rule is shown. In `r4`, the `void` expression in the LHS must be at the same position as in the corresponding agent on the RHS, to signal the creation of a new agent instance. This convention not only serves readability, but also prevents mistakes, when formulating a rule. For example, the unintentional omission of a parameter on the LHS or RHS.
A deletion behaves inversely to creation, as illustrated by `r5`. The to be deleted agent is marked by a `void` in the agent pattern on the RHS. If the rule is then applied to a subgraph matching the LHS, the agent instance is removed from the model.

Listing 3.3: Additional Site States

```
rule r3_1 {B(z{p}[?])}      -> {B(z{q}[?])} @ [1]          //Wild card
rule r3_2 {B(z{p}[bound])} -> {B(z{q}[bound])} @ [1]      //Bound-To-Any
rule r3_3 {B(z{p}[A.x])} -> {B(z{q}[A.x])} @ [1] //Bound-To-Any-Of-Type
```

As opposed to previous examples, link states do not always have to precisely define both ends of a link. They can also be ambiguous, which can be expressed by using a question mark (`?`), as shown in listing 3.3 rule `r3_1`. This is useful when looking for an agent, with a site in a certain state, but the existence of a possible link between this and another site does not matter. Additionally, rule `r3_2` shows the *bound-to-any* expression, denoted by `bound`, which, for example, expresses the requirement for site `z` to posses a link to another site, whose type does not matter. The final expression *bound-to-any-of-type*, demonstrated in rule `r3_3`, refines the previously shown one, by restricting the candidates that are eligible for a connection and is denoted through `<Agent>.<Site>`. For example, the site `z` must have a connection to some other site of type `x`, with its owning agent being type `A`.

Listing 3.4: Observables, Initial and Termination Conditions

```
1  init       i1    10 {A(x[free], y[free])}
2  obs        o1    |{B(z{q}[?])}|
3  terminate  t1    iterations=7000
4  terminate  t2    time=30000
5  terminate  t3    |{B(z{q}[?])}| ==> 45
```

In previous sections, it was just assumed that agent instances existed in the model. However, these must be created first, using the initial conditions, which determine the initial state of the simulation. Such a condition can be defined with the keyword `init`, followed by an identifier for the initial condition, the number of instances to be generated of the following pattern and the agent pattern itself. For example, the initial condition in listing 3.4 line 1 creates 10 agent instances. According to the pattern, these agents have type `A` and their sites `x` and `y` will not be connected to other sites.

The `obs` keyword stands for *observable*, which is used to track agent patterns of interest, so that statistics about their population can be kept for later output and analysis. As illustrated by listing 3.4 line 2, in order to trigger an observation of a pattern the `obs` keyword must be used, followed by a unique identifier and the pattern of interest. After completion of the simulation, population statistics for these patterns are marked with the names given to the observables. In the example, the amount of instances of agents `B`, with their sites `z` in state `q`, would be tracked, without checking for existing links to site `z`.

A simulation would run endlessly, if nothing else was specified. Since this makes little sense, termination conditions are introduced. The simulation terminates as soon as these conditions are met. In order to create a termination condition, the `terminate` keyword, followed by a unique identifier must be used. There are three versions of termination conditions.
First, a termination condition can limit the maximum number of iterations a simulation may perform. For example, the termination condition t1 in listing 3.4 would limit a simulation to 7000 iterations.
Second, a termination condition can also specify the simulation time, after which the simulation is terminated. For example, the termination condition t2 in listing 3.4 would limit the simulation time to 30000 milliseconds. However, simulation time does not refer to the program runtime, but the time elapsed within the simulation, which can be faster or slower, depending on how active the modeled system is (see subsection 2.2.1).
The third and final criterion terminates the simulation, when a maximum number of occurrences of a certain agent pattern has been reached. This can be defined as shown in listing 3.4 line 5, by specifying a pattern followed by a subsequent definition of an upper bound. As a note, all different termination conditions can be used simultaneously, the first one that is met leads to termination.

Since the same agent patterns within a model, written in this DSL, may occur repeatedly, e.g., in observables, termination conditions or rules, `variables` were introduced, to reduce redundancies. For this purpose, the keyword `var` can be used to define variables. These variables may contain numbers, such as integers or floats as well as agent patterns. Thus, redundancy can be avoided and repeating patterns have to be defined only once. After that, variables containing patterns can simply be referenced by observables, rules and termination conditions.

As we can see, many of the presented functions, attached to certain keywords, have syntactic limitations, e.g., unique identifiers. These restrictions are automatically checked, while the model is written in the editor, which is a feature of Xtext and was an additional reason to choose this tool. Furthermore, semantic highlighting is supported, which indicates semantic errors during input, e.g., forgetting a `void` expression in an agent pattern, or entering a negative value as a rule application probability. In addition, scoping is performed in the background, which ensures that only variables, agents, sites, etc. that have previously been defined, can be used in agent patterns. The user will automatically be notified, if a referenced model element is not present. This also allows a limited automatic code completion, i.e., suggestions for the user about possible variables or keywords, which could be used at the moment.

The Reaction Rules DSL is a self-contained plug-in for Eclipse based on Xtext, which can be installed and used independently of the simulation framework. An editor with syntax and semantic highlighting is provided, which checks scoping and automatically generates an EMF compliant model as an XMI file, in addition to the textual representation, when saving. This model can then be used to initialize a simulation in this framework.

3.3 Simulation

This section presents the actual core of the simulation framework, the Simulation module itself. Therefore, the following subsection 3.3.1 describes how agents, mentioned in the previous subsection, are instantiated. Furthermore, subsection 3.3.2 specifies which settings have to be configured, before the simulation can start and which packages are involved in the Simulation module as a whole. Finally, an overview of the simulation process is given in subsection 3.3.3 and each step of the simulation is explained.

3.3.1 Reaction Container Model

As discussed in the DSL subsection, a simulation must have a model that describes the properties and the number of simulation entities as well as their possible interactions. For this reason, the Reaction Rules DSL was introduced, which is a text-based tool and is used to describe simulation models. Given such a model, one important necessary preparation that has to be done before running the simulation, is the generation of agent instances according to initial conditions.

The simplest approach would be to make copies of the agent patterns, defined in the initial conditions, as often as necessary. However, it is not that simple for several reasons. For example, in these agent patterns connections between sites are only assigned by indexes, i.e., ordinary integers that do not carry any information, to which sites they belong to. This would not only make pattern matching more difficult, but also makes rule application through graph transformation inefficient. Furthermore, the model carries a lot of information, e.g., the previously mentioned initial conditions, variables, etc., which are no longer important in the running simulation. However, the main reason not to follow the simple approach comes from the experience that generating models, with large numbers of agent instances, can take some time. A generation would be triggered, every time a valid model is saved in the editor of the DSL. This could be prevented through a different implementation of the model generator and by dividing the model generation into two steps. The first part would be triggered after the creation of the model in the DSL editor and would not yet have any instances of the agents. Followed by a second model generation, which would then insert the necessary agent instances into the model. However, this approach poses a huge problem because the synchronization between the textual model definition in the DSL and the representation of the model in the XMI file would be lost.
In order to avoid these problems, a second metamodel is defined, in addition to the DSL metamodel, whose instances serve only as containers, in which simulations run. The Reaction Container metamodel is shown in figure 3.2, it carries the agent instances, as defined in the initial conditions, inside a container object. As shown in the figure, agent instances are referred to as *SimAgents*, sites as *SimSites* and states as *SimSiteStates*. These names are chosen to clarify that these are instances of agents and are used in the simulation. Connections between agents, via their sites, are established by references to *SimLinkState*

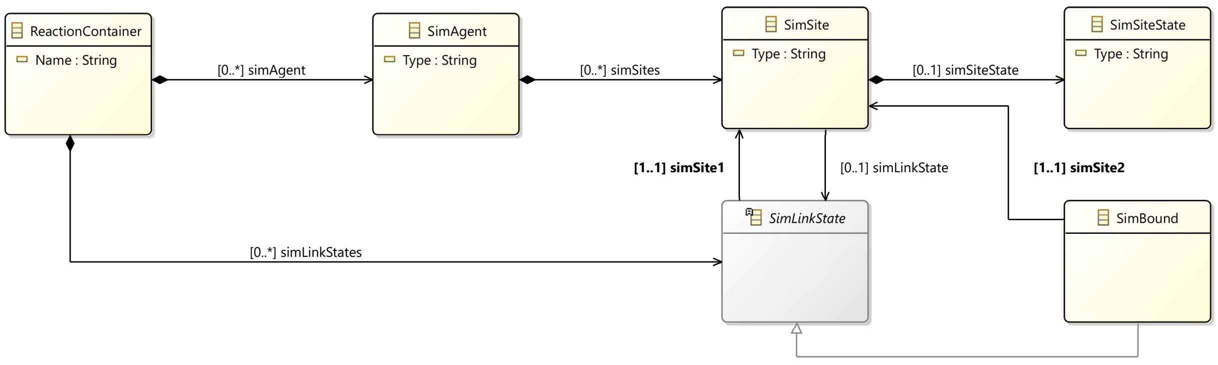


Figure 3.2: Reaction Container Model

objects. This simplifies the pattern matching later on, since only possession and equality of SimLinkState objects must be checked, in order to find connected agents.
The presented metamodel is deliberately kept very simple, but it leaves room for expansion. For example, it is possible to add other link types, through implementing the SimLinkState interface.

3.3.2 Simulation Setup

Before the details of the simulation process are discussed in detail, it is important to first look at the involved modules and the necessary steps to configure them. Figure 3.3 provides an overview of the main modules involved in the simulation.

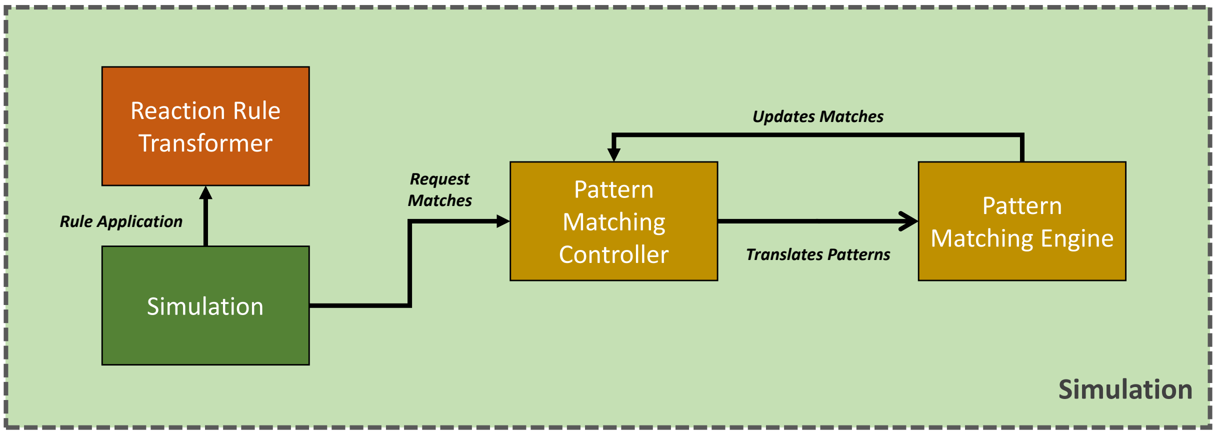


Figure 3.3: Simulation Modules

The first module is the *Reaction Rule Transformer*, which contains all classes needed to apply rules formulated in the DSL to the model, during simulation. The next module is the Pattern Matching Controller, which ensures that patterns, specified in the rules, are converted in such a way that general-purpose pattern matching tools can process them. The *Pattern Matching Engine* module works in tandem with the PMC and contains classes that wrap the pattern matching tools. Additionally, it processes matches in such a way, that they can be used by the PMC. Simulation, the main module, implements a stochastic simulation, according to the algorithm described in subsection 2.2.1. In addition, the

Simulation module contains classes that implement statistics output, termination conditions, and configuration of the simulation.

The setup itself is divided into two phases. First, the configuration of simulation features, via the Simulation Configurator. During this, the Reaction Rules model, to be loaded, and the Reaction Container model, to be generated, are set. This is accompanied by a definition of the persistence solution used to load the model. The default variant is using XMI files, supported by the EMF framework. Alternatively, an EMF compliant graph-based database, called NeoEMF[7], can be chosen as well. Furthermore, the pattern matching tool is selected, which can either be Viatra or Democles. Finally, the implementation variant of the pattern matching controller is chosen. This can be the simple variant, which does not preprocess patterns and only translates them for the pattern matcher. Alternatively, an extended PMC variant can be picked, which further processes patterns, with the aim to improve pattern matching performance. This subject is explained in detail in subsection 3.4
The second phase starts after the configuration with a newly created simulation object. During its initialization, the simulation object loads the specified Reaction Rules model and the corresponding Reaction Container model. Furthermore, all rules are extracted and passed to the transformation module, which can then initialize its graph transformations. In addition, all patterns are extracted and transferred to the PMC, which then supplies them to the Pattern Matching Engine module and initializes the pattern matcher with the received Reaction Container model. Finally, all observables and termination conditions are generated, to enable the tracking of statistics on pattern populations and the automatic termination of the simulation.

3.3.3 Simulation Execution

The simulation implements the concept of Gillespie's algorithm, presented in subsection 2.2.1, and consists basically of a single loop.

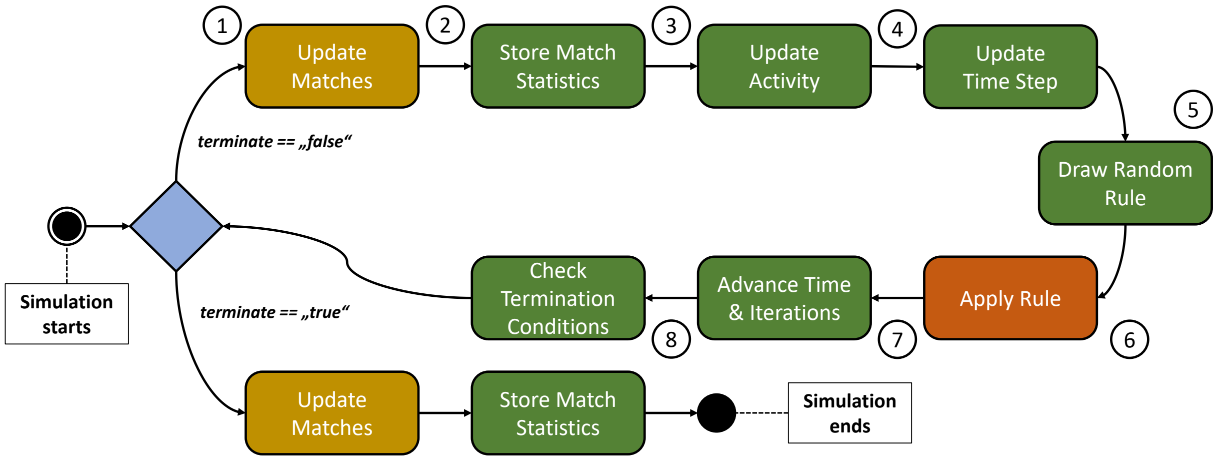


Figure 3.4: Simulation Loop

[7] NeoEMF project page: https://www.neoemf.com

At the beginning, as illustrated by figure 3.4, the state of the model is checked. This means, in step 1, the pattern matching engine is ordered to update its internally stored matches to patterns, specified in the Reaction Rules model. The EMF framework ensures that a notification is automatically sent to the pattern matcher, each time the model is changed by the application of a rule. Following that, all registered changes, applied to the model in the last iteration, are taken into account and all matches are incrementally updated, as described in subsection 2.3.3.
Following that, in step 2, the simulation's state, stored in a state object, is transferred to an instance of the *SimulationStatistics* class. It checks if observables want to record certain pattern population statistics and queries the active PMC, contained in the state object, for the match counts corresponding to patterns defined by observables. These statistics are stored in addition to the current simulation time and can be displayed as an xy-plot, when the simulation has finished.
After this, in step 3, the activity of the individual rules and, thus, the activity of the entire system can be determined according to the first step of Gillespie's algorithm.

$$a_\mu = h_\mu c_\mu \,, \qquad \mu \in [1, ..., M] \tag{10}$$

The activity a_μ can be calculated using equation 10 and requires the number of matches h_μ, provided by the pattern matcher, and the static rule application probability c_μ of a rule R_μ. The application probability indicates how likely it is that a rule will be applied to a found match of its LHS. Consequently, the activity of a rule R_μ is a measure of how likely it is that a rule will be selected and applied, given the number of found matches and its static application probability c_μ.

$$a_0 = \sum_{v=1}^{M} a_v = \sum_{v=1}^{M} h_v c_v \tag{11}$$

The activity a_0 of the system is determined by equation 11. Since a_0 is the sum of all rule activities, it can be used to discern whether the simulated biochemical system is particularly active or not.

$$\tau = \left(\frac{1}{a_0}\right) ln \left(\frac{1}{r_1}\right) \tag{12}$$

In Step 4, the time interval τ can now be determined through equation 12, using the previously calculated system activity a_0 and a random number r_1. When looking at this, we can see that the length of the time interval in Gillespie's algorithm depends on how active the system is. This makes sense because for occasions, where many reactions happen in the system in a small amount of time, the temporal resolution must be high in order to simulate all of them. If, on the other hand, the system is very slow, then it would be rather unfavorable to make many small simulation steps, in which nothing would happen.
Once the time step and the system activity have been determined, a rule R_μ can be selected in step 5. This does not happen arbitrarily, but according to a distribution

density function, as postulated by Gillespie. The rule is selected in such a way that the equation 13 is satisfied.

$$\sum_{\nu=1}^{\mu-1} a_\nu < r_2 a_0 \leq \sum_{\nu=1}^{\mu} a_\nu \tag{13}$$

If we consider this equation, we can see that with the help of a random number r_2, between 0 and 1, as well as the system activity, a new number is determined, which lies between 0 and the system activity a_0. Thinking about the way the system activity is composed, namely from the sum of the rule activities, it becomes clear that the previously selected number is located in an interval defined by a rule's activity a_μ. The drawn random number then determines the rule through the corresponding interval it fits into. Thus, the probability that a rule is selected is proportional to its activity. This in turn implies that rules, whose preconditions have no matches, can never be selected because their activity amounts to zero.
In step 6, the identifier of the selected rule is transferred to the Reaction Rule Transformer module, along with a randomly selected match from the set of all matches to a rule's LHS. In the module, a graph transformation, as presented in subsection 2.3.2, is performed, which means that the match to the LHS of the rule is changed, such that it corresponds to the RHS of the rule. This could, for example, result in states and link states of sites being deleted or moved and agent instances being created or deleted in order to transform the match according to the RHS of the rule. Since Reaction Container models are all based on EMF, the EMF framework takes care of propagating changes applied to the model, i.e., modifications made to entities referenced by a match, by sending notifications. As indicated in the setup paragraph, the receiver of such notifications are general-purpose pattern matching tools, wrapped by the Pattern Matching Engine module.
Following the rule application, in step 7, the simulation time of the simulation state is updated, i.e., incremented by the value of the current time step. In addition, the state's iteration counter is increased by 1.
In the final step of the loop, it is checked if one of the termination conditions, previously defined in the Reaction Rules model, is satisfied. For this purpose, the *SimulationState* object is given to an instance of the *SimulationTerminationCondition* class. This object checks whether a possible time or iteration limit has been reached. If this is not the case, it is checked whether limits for pattern populations have been defined. If so, the PMC is asked for the corresponding match counts for these patterns. When one of the defined termination criteria is fulfilled, the main loop of the simulation is exited. Otherwise, the simulation is resumed at the first step.
When leaving the simulation loop, the pattern matcher is ordered to update all matches, so that at the end an up-to-date output of the match counts of different patterns is possible.

As we can see, the basic concept of the simulation is kept quite simple and implements the ideas of Gillespie. The difference to the other approaches, presented in section 5, also based on Gillespie's algorithm is that in this approach, the tracking of all possible matches to agent patterns and especially their modifications is not performed through book keeping. Instead, general-purpose pattern matching tools are used. How these tools are integrated as well as the necessary steps needed for these tools to be able to handle agent patterns, is explained in the following subsection.

3.4 Pattern Matching

As we could see in the previous subsection, it is of central importance to be able to find all matches to the LHS of rules, defined in the DSL, during the simulation. The number of these matches is important in order to calculate, for example, the activity of a rule. Additionally, an actual instance of a rule's precondition, i.e., a match, is required so that a graph transformation, described by a rule, can be performed. As indicated at the end of subsection 3.3.3, other simulation tools that also simulate biochemical processes, with the help of rule-based modeling and stochastic simulation, often use proprietary solutions to find matches on rule preconditions. In most cases these solutions are not described in detail and work as black boxes in the background. In the framework presented in this thesis, general-purpose pattern matching tools are used for the purpose of finding matches. These are characterized by their high degree of expressiveness, which means that these tools can be used to describe anything, from simple to very complex patterns, within all conceivable problem domains. However, this generic nature implies that agent patterns, described in the DSL from subsection 3.2, cannot be used by the pattern matchers without any preprocessing. These patterns have to be translated into patterns that can be understood by a general-purpose pattern matching tool. This process could be a translation into a proprietary DSL, as it is the case with Viatra. On the other hand, such a translation could also imply that the patterns are translated directly into the tool's own patterns, conforming to some metamodel, as is the case with Democles.
Since a translation step is required, the so-called Pattern Matching Controller was introduced as part of this work. One of the module's tasks is to convert patterns that were defined using the Reaction Rules DSL in such a way that they are usable by the respective pattern matching tool. In addition, matches returned by the pattern matcher must be converted to a more generic representation, since it should be of no concern to the simulation from which type of pattern matcher these matches came from. This prevents the creation of specialized solutions for each employed pattern matching tool and generally simplifies switching between different pattern matchers. Finally, there is another motivation behind the PMC, namely to preprocess patterns, if necessary, before converting them in order to make the process of pattern matching, using general-purpose pattern matchers, more efficient and, hence, to improve performance. An example of such preprocessing, which is already built into this framework, is given in subsection 3.4.4. However, the following subsection will first explain how to convert patterns into the Viatra pattern language. Following that, in subsection 3.4.2, the necessary steps for the conversion into Democles patterns are explained.

3.4.1 Viatra Patterns

As mentioned before, the Viatra pattern matching tool comes equipped with its own DSL. This allows the specification of patterns, for which matches are to be found in a model. The interpretation of these patterns is performed by the Viatra framework. This means, that the user does not have to worry about creating pattern models, which have to conform to the internally used Viatra pattern metamodel.

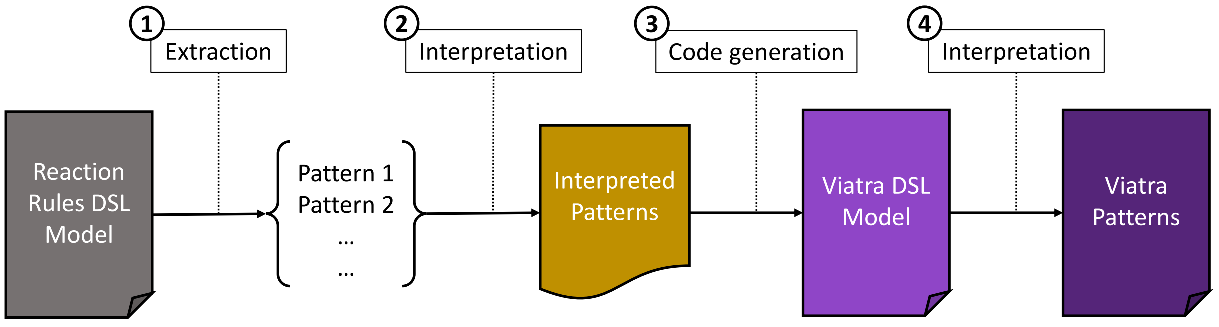


Figure 3.5: Viatra DSL Translation

The work flow from Reaction Rules DSL patterns to an initialized Viatra-based pattern matcher is as follows: In the first step, all agent patterns are extracted from a model created with the Reaction Rules DSL, as shown in figure 3.5. As described in subsection 3.3.2, this step is performed during the setup phase, immediately before the simulation starts. In step 2, these patterns are interpreted and in step 3, converted into the syntax of the Viatra DSL, using a code generator. The model described through the Viatra DSL is interpreted by the Viatra framework, as mentioned above, and in the final step, Viatra patterns are generated from it. Using these patterns, a new pattern matcher is created, which can be initialized with a model that conforms to the Reaction Container metamodel.

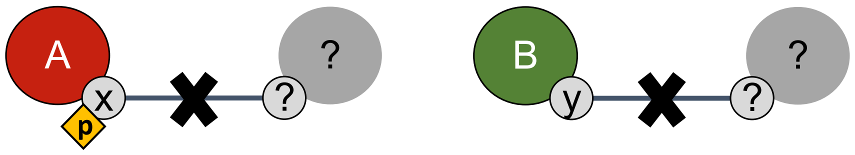


Figure 3.6: Pattern Translation – Example Pattern

How agent patterns, written in the Reaction Rules DSL, can be converted to patterns conforming to the Viatra DSL, is explained in the following paragraph, using an example. Figure 3.6 shows a simple pattern, which requires an agent of type `A`, whose site `x` is in a state `p` (orange diamond) and is not connected to another site. In addition, this pattern requires an agent of type `B`, whose site `y` can be in any state and must not be connected either. Listing 3.5 line 2 shows this pattern in the syntax of the Reaction Rules DSL.

Listing 3.5: Pattern Translation – Reaction Rules DSL Pattern (1)

```
1 // Reaction Rules DSL
2 {A(x{p}[free]), B(y[free])}
```

Listing 3.6 shows the much longer Viatra code, which corresponds to our short example in listing 3.5. Looking at the Viatra code, we can see that the pattern is divided into a signature and a body. In the signature, a unique identifier is assigned to the pattern and global context nodes are defined. Global context nodes serve a similar purpose like parameters that are passed when a function is called. They are available in the body as constant variables and can be referenced. In listing 3.6 line 2, these context nodes represent two SimAgents from the Reaction Container metamodel, which can be referenced using the identifiers `A` and `B`.

Listing 3.6: Pattern Translation – Viatra-DSL Pattern

```
 1  // Viatra—DSL
 2  pattern p1 (A : SimAgent, B : SimAgent) {
 3          // Define agent types
 4          SimAgent.Type(A, ”A”);
 5          SimAgent.Type(B, ”B”);
 6          // Define site types & states
 7          SimAgent.simSites(A, A_x);
 8          SimSite.Type(A_x, ”x”);
 9          SimSite.simSiteState.Type(A_x, ”p”);
10          SimAgent.simSites(B, B_y)
11          SimSite.Type(B_y, ”y”);
12          // Define link states
13          neg find support_pattern(A_x);
14          neg find support_pattern(B_y);
15  }
```

Line 4 in the body ensures that a SimAgent instance, referenced through identifier `A`, has type `A` as well, by using an attribute constraint. This means, that for a correct match only SimAgent instances are allowed, whose string attribute, named `Type`, has the value `"A"`. Analogous, a similar constraint is applied to SimAgent `B` in line 5.

For the next steps local context nodes are required. These nodes represent context that must exist in addition to the signature nodes, for a match to be valid. For example, sites are not part of the signature in the Viatra pattern, but according to the pattern in listing 3.5, must exist and must have a certain type. In order to ensure that this required context exists, local context nodes describing these sites are defined. For this purpose, local context nodes for SimSites are created in line 7 and 10, using a constraint, which ensures that a site belongs to a certain agent. For example, this would check if a reference to a SimSite instance is located in the site container attribute of SimAgent `B`. In lines 8 and 11, site types are determined analogously to the definition of a SimAgent type. In the case of SimAgent `B`, this would be the type `y` and for the SimSite belonging to `A`, the type `x` is required.

A local context node is also created for the state of a site, expressed through a SimSiteState object. In line 9, a constraint ensures that this instance also belongs to SimSite `x` of SimAgent `A` and has the type `p`.

Finally, it is ensured that site `x` and site `y` are free, which means that they do not have existing connections to other sites. A so-called support pattern is used for this, since the absence of context nodes can not be formulated in the Viatra DSL. A support pattern is called from within a different pattern's body and can receive context nodes as parameters, similar to a function call. The returned match set of the support pattern, is then used as context by the calling pattern.

Listing 3.7: Pattern Translation – Viatra-DSL Support Pattern

```
pattern support_pattern(site : SimSite) {
        SimSite.simLinkState(site, _);
}
```

Such a support pattern is shown in listing 3.7 and requires a SimSite as a global context node. The attribute constraint in line 2 demands that a SimSite instance must have a SimLinkState instance, in order to match this pattern. The underscore represents a wild card and expresses that any type of SimLinkState object is allowed. In listing 3.6 lines 13 and 14, this support pattern is called and its returned matches contain all SimLinkState objects referenced by the given SimSites with type `x` or type `y`, respectively. The negation, expressed through the `neg` keyword in both lines, has the effect that this pattern is only matched, if said sites do not have references to SimLinkState objects.
If we want the opposite, i.e., both SimAgents should be connected to each other, we must create local context nodes representing SimLinkStates, for both SimAgents `A` and `B`. These two local context nodes can then be checked for equality. If both are equal, it is certain that the instances of `A` and `B` are connected to each other, at sites `x` and `y` through the same SimLinkState instance.

Listing 3.8: Pattern Translation – Reaction Rules DSL Pattern (2)

```
{A(x[free]), A(x[free])}
```

One problem, however, has not yet been taken into account, that of injectivity. When looking at listing 3.8, we can see that, this time, the second required agent also has the type `A`, as opposed to listing 3.5. If we would translate this into a Viatra pattern naively, the pattern matcher would return matches containing the same instance of SimAgent `A` twice. This means, that this instance was selected for both, the first and the second global context node, which does not make sense when we look at what the pattern implies, i.e., two different proteins of the same type. In order to avoid this problem, so-called injectivity constraints are introduced. These are automatically inserted into the pattern, if SimAgents of the same type are required in the global or local context. This can easily be realized by using a constraint, demanding two SimAgents of the same type being unequal. If two SimAgents are equal, the pattern containing the injectivity constraint is not matched.

Using the above presented schema, any pattern formulated in the Reaction Rules DSL can be translated into the Viatra DSL. As a remark, a detailed explanation for the translation of *Bound-To-Any* and *Bound-To-Any-Of-Type* link states is omitted, since it would only lead to the reiteration of previously described steps, in a slightly modified form. This would not contribute to the understanding of the translation process and only increase the complexity of the given examples.

3.4.2 Democles Patterns

Unlike Viatra, the Democles pattern matching tool does not include a DSL, which could be used to define patterns textually. Democles requires that patterns are compiled directly from the components defined in the Democles pattern metamodel. In the framework presented in this thesis, a pattern compiler, written in Java, constructs patterns that conform to the Democles pattern metamodel and semantically correspond to the patterns defined in the Reaction Rules DSL.

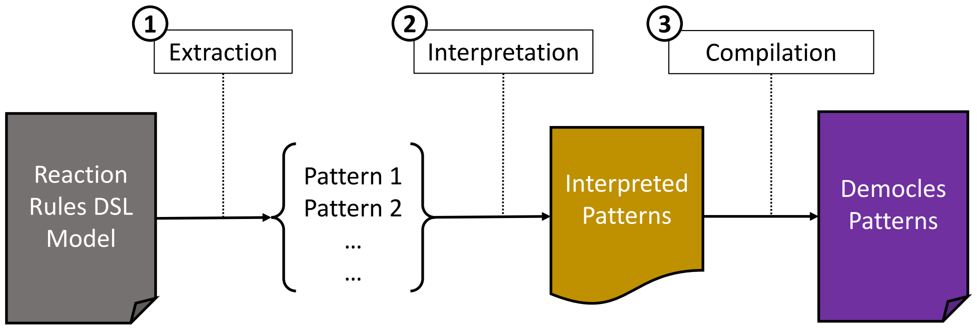


Figure 3.7: Democles Pattern Translation

The work flow from the Reaction Rule model to the initialized Democles-based pattern matcher is shown in figure 3.7. In the first step, patterns are extracted from a model created with the Reaction Rules DSL, during the setup phase of the simulation. As a second step, these patterns are interpreted and unlike Viatra not converted to another DSL, using a code generator. Instead, they are directly compiled into Democles patterns. With these patterns and a Reaction Container model, a Democles pattern matcher can be created and initialized.

In the following paragraph, the process of converting Reaction Rule patterns to Democles compliant patterns is explained, using the example shown in figure 3.6 and listing 3.5.
Initially, a *Pattern* object must be created and a unique identifier is assigned, analogous to Viatra patterns in subsection 3.4.1. This Pattern object has a signature, which contains the global context nodes and a body, describing the requirements of the pattern.
To define a signature, a *PatternSignature* object must be created, to which all context nodes corresponding to the agents in the agent pattern are added, one after the other. Such a context node is represented by an *EMFVariable* object, which has a unique name and knows the type of the context node. For example, `A` as variable name and SimAgent as type, in the case of agent `A`. Semantically, this has the same meaning as the expression in line 2 in listing 3.6.
The body of the pattern is defined through a *PatternBody* object. First, the types of the agent context nodes are defined, similar to a Viatra pattern body. An *Attribute* object ensures that the context node, here an instance of a SimAgent, has a type attribute. This is a constraint that does not check the value of the attribute, but its existence and data type. When we consider the example, this would be a check for the existence of an attribute with the name `Type` and its data type `String`, for SimAgent `A` as well as for SimAgent `B`. In Democles, the check for a certain attribute value is then performed with a *RelationalConstraint* object. As parameters, the RelationalConstraint object receives the type of comparison, e.g., *Equal* or *Unequal*, a reference to the attribute, in which the value to be compared is located, and the value that must be checked. For example, the

value `"A"` in the case of SimAgent `A`. This procedure, semantically, corresponds to the line 4 and 5 in listing 3.6.
The definition of a SimAgent's SimSite type, is almost analogous to the definition of the agent type. The difference is that the existence of a reference to a SimSite object, in the SimSite's container attribute of a SimAgent object, must first be ensured. For this purpose, a *Reference* object is used in Democles, which receives the type of the container attribute, the SimAgent itself and a local context node, representing a SimSite. This reference constraint checks, if a reference to a SimSite object exists in the container attribute of a SimAgent object. The required site type, is then ensured the same way the agent type is checked. Again, an Attribute object is created, which receives the data type and the name of a SimSite's type attribute, in this case `Type` and `String`. Then the value of the attribute `Type` is checked, with the help of a RelationalConstraint object and compared to the required value of the attribute. For example, `x` is the value for the site type, in the case of agent `A` from listing 3.5. Equivalent to this, are lines 7 through 8, and 10 through 11 in listing 3.6.
The site state is determined analogously to the procedure, just described. When the reference constraint is created, it is not checked if a SimAgent has a reference to its SimSite. Instead, the constraint checks if a SimSite has a reference to a SimSiteState object. Then, the name and data type of the `Type` attribute as well as its value is defined, with the help of the attribute and relative constraints. This corresponds to line 9 in listing 3.6, in the Viatra DSL. For example, site `x` should have a state `p`, in case of agent `A`.
In order to ensure that SimAgents do not have an existing connection at their SimSites, a support pattern has to be used for Democles as well. The support pattern is designed to find all SimLinkState objects, which belong to SimSite context nodes having a certain type. This is done using a Reference object, the same way as checking the existence of SimSites in a SimAgent. The resulting constraint, then ensures that an instance of a SimSite has a reference to a SimLinkState object. This support pattern, similar to the one in the previous subsection, is called and negated. As a consequence, only SimSite objects that do not have a reference to a SimLinkState object are allowed. This ensures that SimAgents are not connected at SimSites.
If, on the other hand, we want SimAgents to have a connection at their SimSites, we can proceed in the same fashion as when defining sites and states. This means that for SimAgents a reference constraint is used to check, if their SimSites have references to SimLinkState objects. These SimLinkStates are then checked for equality, with a RelationalConstraint.
Using Democles, injectivity constraints must be inserted as well, to prevent multiple usage of the same SimAgent object in a match. This is done at the end of the pattern body creation. For this purpose, a relational constraint is introduced for each pair of context nodes representing a pair of SimAgent objects with the same type. This constraint does not check for equality of the given context nodes as before, but for inequality.

As in subsection 3.4.1, the description of the conversion for other link states is omitted as well, since this would not present any new Democles functionality. In fact, it merely repeats the steps already explained.

As we can see, the process used to translate Reaction Rule patterns to Democles patterns is shorter, but a the same time more complicated, compared to Viatra. The advantage, however, is that the step of code generation and reinterpretation through the Viatra framework is omitted.

3.4.3 Disjunct Sub-Patterns

With the previously described methods Reaction Rules DSL patterns can be translated directly into the appropriate form, required by the general-purpose pattern matching tools. For these patterns, the corresponding pattern matcher finds all matches in a given model. This set of matches is then used to determine the probability with which a rule is applied, as described in 3.3.3. For this, the PMC module has to translate the patterns, initialize the pattern matcher, update the match set of a rule, if necessary, and output the number of matches.
However, certain patterns have turned out to be problematic, making it necessary to extend this process. These problematic patterns all share one property, namely the fact that some of the described agents are not linked, through their sites, to other agents in the pattern. This means, that the graph, which represents such a pattern, is not fully connected. Therefore, the graph can be divided into a set of unconnected subgraphs, which in turn represent sub-patterns of the given pattern. Nodes in this kind of graph represent agents and edges represent links between sites.

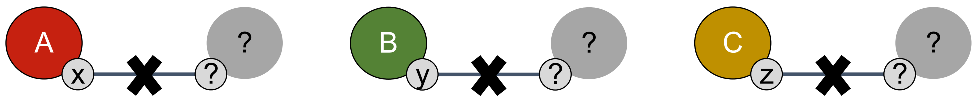


Figure 3.8: Disjunct Sub-Patterns – Example Pattern

Figure 3.8 shows an example of such an unconnected pattern, henceforth, called disjunct pattern. In this example, three agents `A`, `B` and `C` are defined, whose respective sites `x`, `y` and `z` are free and, therefore, represent three independent sub-patterns. Here, it is assumed that a Reaction Container model contains 4 SimAgents for each type `A`, type `B` and type `C`, whose SimSites do not contain a reference to SimLinkStates. This means, that a pattern matcher will find 4 matches for each sub-pattern. In order to determine the entire match set for the pattern from figure 3.8, the pattern matcher basically tries all possible combinations of matches on the sub-patterns and remembers the combinations that do not contradict any defined constraints. For example, the number of all ordered pairs, which can be constructed from matches on sub-pattern `A` and `B`, corresponds to the product of the number of matches on `A` and `B`. This is basically the product of two sets, which is also called cross product in set theory and is defined according to equation 14.

$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\} \tag{14}$$

$$X_1 \times \ldots \times X_n = \{(x_1, \ldots, x_n) \mid x_i \in X_i \ \forall \ i \in \{1, \ldots, n\}\} \tag{15}$$

Looking at the example from figure 3.8, we can see that we do not only need the cross product of match sets `A` and `B` but also of `C`. This is called an *n-ary Cartesian product*, which is defined in equation 15 and produces the set of all nested ordered pairs that can be formed from n sets. In the case of this example, the number of matches in the whole match set is be given through $|A \times B \times C| = |A| \cdot |B| \cdot |C| = 4 \cdot 4 \cdot 4 = 64$

If we think about the potentially massive numbers of agents, which may appear in the simulation of biochemical processes, it becomes clear that match counts can explode very quickly. As a result, the matching process may take an unreasonable amount of time, since all combinations have to be checked. Additionally, due to the way the Rete-Networks, used in pattern matchers (see subsection 2.3.3), manage matches, this explosion in match occurrences will lead to a great increase in memory consumption.

3.4.4 Hybrid Pattern Matching

As described in the previous subsection, disjunct patterns can lead to a major performance problem. This is especially problematic because this pattern type is very common in the problem domain biochemistry. After all, we typically want to connect molecules that are not yet connected with each other. To find such unconnected molecules, we need to formulate said problematic patterns. Consequently, it is essential to avoid the explosive increase of matches to such patterns.

The idea is to divide disjunct patterns into their constituent sub-patterns. Then, translate these sub-patterns and initialize a pattern matcher. To calculate the match count of an original disjunct pattern, we use the match count of its sub-patterns. Since this method does not purely rely on the power of general-purpose pattern matching tools, but instead, introduces a preprocessing step to improve pattern matching performance, it is, henceforth, called *hybrid pattern matching*. This hybrid approach will prevent the usage of disjunct patterns with pattern matchers and, thus, remove the explosive increase of match occurrences.

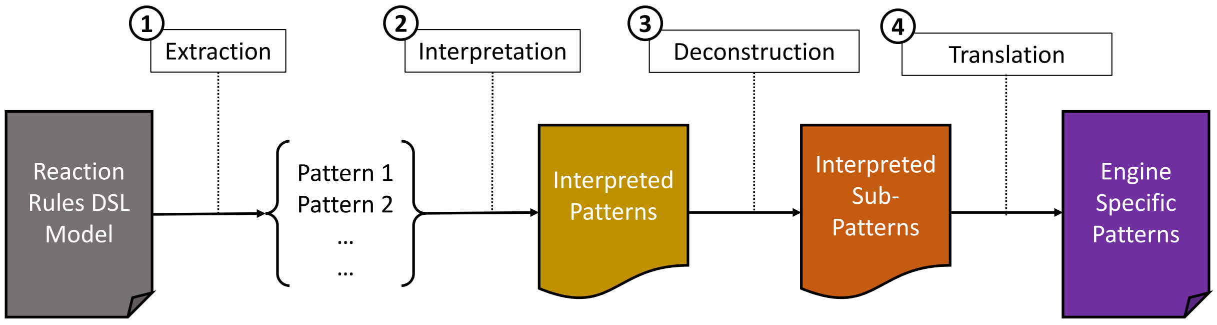


Figure 3.9: Hybrid Pattern Translation

As depicted by figure 3.9, an additional step is inserted into the pattern translation process, making use of the abstraction layer provided by the PMC module. This additional step preprocesses patterns in the setup phase of the simulation. During this preprocessing, the PMC splits original patterns into its constituent sub-patterns. Furthermore, it stores a mapping from the original patterns to its sub-patterns and, thus, enables a mapping of sub-pattern match counts to their original patterns. As a result, the match count of an original pattern can be determined, using the n-ary Cartesian product (see equation 15) and the mapped match counts of its sub-patterns.

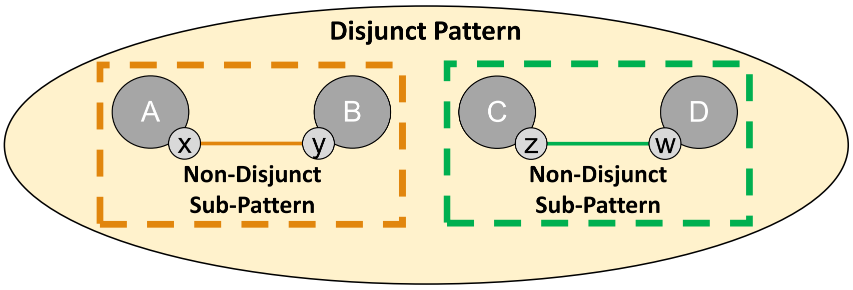


Figure 3.10: Hybrid Approach – Disjunct Pattern (1)

For this procedure to work, all sub-patterns that constitute the original patterns must be determined. A useful characteristic, which helps us to do this task, is the fact that patterns can be represented by graphs. Hence, disjunct patterns can be described by a set of non-disjunct subgraphs. When we look at the example pattern in figure 3.10, it becomes clear what is meant by that. In the shown pattern, agent `A` and agent `B` belong to the left subgraph (orange) and agents `C` and `D`, to a second subgraph on the right (green). In these graphs, agents `A` to `D` correspond to nodes and link states, connecting the sites to edges.
Algorithm 1 shows how subgraphs are discovered in this thesis. The procedure is basically a tree traversal, not unlike Prim's or Kruskal's algorithm, used to find minimal spanning trees. The major difference, is that finding the minimal spanning tree is not important. Instead, finding any tree that connects all nodes, i.e., agents in a subgraph, is sufficient. For this purpose, a node of the node set, representing a copy of the original pattern graph, is removed in each iteration of the main while-loop, in line 4, and added to an empty set. Then, all outgoing edges, described through link states of an agent's sites, are extracted and packed into a list. If this list is empty, the current node does not have outgoing edges, which means that the current subgraph consists only of this singular node and is, therefore, completed. In the inner while-loop, in line 9, edges from the list are popped, i.e., removed from the top in each iteration. If the current edge is not of type `Bound`, it does not connect two nodes, and is, therefore, discarded. After that, the target node is selected, i.e., the node that is the target of this edge, with its source being the current node from line 5. This node is then removed from the set of pattern nodes and its extracted outgoing edges are added to the current list of edges. In line 18, the target node is finally added to the current set of sub-pattern nodes. The inner loop terminates, when there are no edges left in the outgoing edges list. After the inner loop is finished, the completed sub-pattern, represented by a set of nodes, is added to the list of sub-patterns. The outer loop terminates, when there are no nodes left in the set that contained all pattern nodes. When the outer loop terminates, the list of patterns is returned.

$$N = |X_1 \times \ldots \times X_n| = \prod_{i=1}^{n} |X_i| \tag{16}$$

Using match counts $|X_i|$ of sub-patterns, gained through the described algorithm, and

Algorithm 1 Finding Sub-Patterns using a Spanning Tree Algorithm

```
 1: function SPLITPATTERN(setOfNodes)
 2:     listOfPatterns ← ∅
 3:     pattern ← $copy$(setOfNodes)
 4:     while $notEmpty$(pattern) do
 5:         currentNode ← $pop$(pattern)
 6:         currentPattern ← {currentNode}
 7:         outgoingLinks ← $extractOutgoingLinks$(currentNode)
 8:         if $empty$(outgoingLinks) then
 9:             $addToList$(listOfPatterns, currentPattern)
10:             $continue$
11:         while $notEmpty$(outgoingLinks) do
12:             currentLink ← $pop$(outgoingLinks)
13:             if $equals$(currentLink.type, ”Bound”) then $continue$
14:             if $unequals$(currentLink.source, currentNode) then
15:                 candidateNode ← currentLink.source
16:             else
17:                 candidateNode ← currentLink.target
18:             $removeFromSet$(pattern, candidateNode)
19:             $addAllToList$(outgoingLinks, $extractOutgoingLinks$(candidateNode))
20:             $addToSet$(currentPattern, candidateNode)
21:         $addToList$(listOfPatterns, currentPattern)
22:     return listOfPatterns
```

applying equation 16, derived from the n-ary crossproduct, in principle, the total match count N of a disjunct pattern can be determined. However, there is still one case that needs to be considered in order for this approach to work.

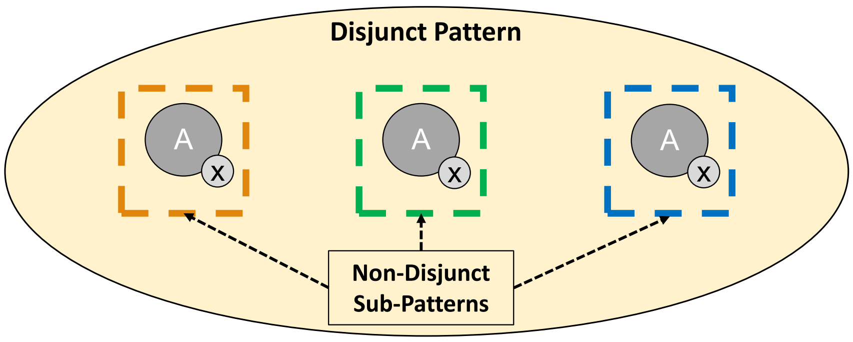


Figure 3.11: Hybrid Approach – Disjunct Pattern (2)

Figure 3.11 shows a pattern, which would generate the wrong match count, if equation 16 was used. An injectivity constraint is required for this pattern, so that the same SimAgent instance with type `A` is not used 3 times in the same match. Consequently, this means that such a constraint subsequently reduces the match count, by the number of matches that violate the constraint. This circumstance is not considered in equation 16, which simply represents the n-ary Cartesian product. If we think about what the cross product represents, namely the set of all ordered pairs of elements that can be constructed from two sets, it becomes clear that an injectivity constraint only removes those pairs, where the same element is used twice.

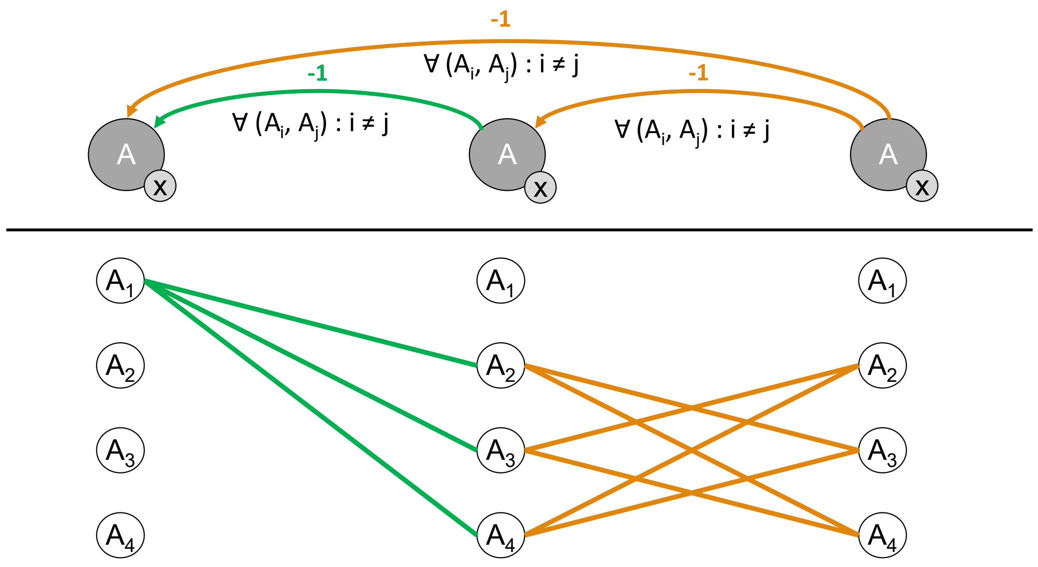


Figure 3.12: Hybrid Approach – Injectivity Constraints (1)

Generalizing this to arbitrarily sized patterns is a little more complicated and is best explained using an example. For this reason, figure 3.12 is provided, which shows the matches found for their respective sub-pattern. SimAgent instances are listed below their matching nodes, representing agents of type `A`. The model, in this example, has 4 SimAgents of type `A`. Since all sub-patterns describe the same agent, each sub-pattern receives the same set, containing 4 matches. The injectivity constraints, which must be satisfied by matches of sub-patterns in figure 3.12, are represented by directed edges between nodes in sub-patterns. Now, we begin to construct pairs from matches of the first two columns, by connecting them through edges (green). Due to the injectivity constraint (green), we may not connect matches that have the same index, meaning, they represent the same instance. The match count until now would be $4 \cdot 3 = 12$, instead of $4 \cdot 4 = 16$, because we had to remove a pair of matches violating the constraint. Next, we connect matches from the third column through edges (orange), forming triplets. In order to satisfy the constraint, no match with the same index may appear twice in a particular triplet. The final match count amounts to $4 \cdot 3 \cdot 2 = 24$, instead of $4 \cdot 4 \cdot 4 = 64$. If this pattern would contain another node, representing an agent of type `A`, the match count would amount to $4 \cdot 3 \cdot 2 \cdot 1 = 24$, with an additional node leading to a match count of 0. Consequently, if there are more sub-patterns requiring an instance of certain type, than there are instances of this required type, no match to the original disjunct pattern can be found.

$$N = \prod_{i=1}^{n} [|X_i| - f(X_i)] \tag{17}$$

Equation 17 is a modified version of equation 16, representing the previously described match count calculation. In this equation, the match count $|X_i|$ is reduced by its constraint factor $f(X_i)$, which is gained through static analysis of sub-patterns, during the setup phase. This factor represents the amount of constraints, shared by a sub-pattern P_i and its predecessor sub-patterns $P_i - 1, ..., P_1$, of the original pattern. For example, in figure 3.12 the second sub-pattern P_2 shares an injectivity constraint (green edge) with its predecessor P_1 and, therefore, has a factor of $f(X_2) = 1$. Whereas, sub-pattern P_1 has a factor of zero because it is the first sub-pattern and, hence, does not have a predecessors to share injectivity constraints with. Finally, sub-pattern P_3 has a factor of $f(X_3) = 2$ because it shares two injectivity constraints (orange edges) with its predecessors. When

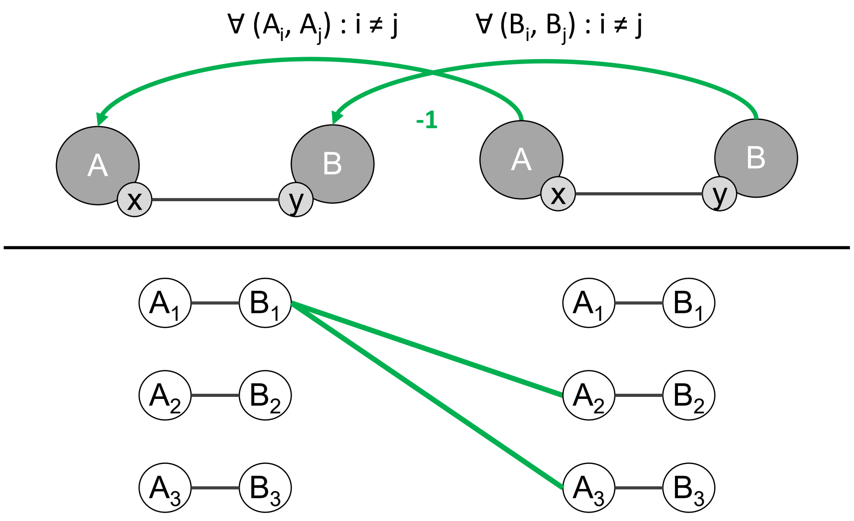


Figure 3.13: Hybrid Approach – Injectivity Constraints (2)

two sub-patterns share more than one injectivity constraint, as depicted in figure 3.13, the match count is still only decreased by one. This stems from the fact that sub-patterns are by design always connected. It leads to the effect that constraints, referring to any of the nodes in the sub-pattern, remove matches that would otherwise match the rest of the nodes in the pattern, without conflicting with their constraints. On the other hand, multiple violations of constraints, through a match of a single sub-pattern, do not result in the decrease of match count by the same amount. For example, the match containing `A1` connected to `B1`, in figure 3.13, would violate two injectivity constraints of the right sub-pattern, under the assumption that the same match has been selected for the first sub-pattern. Nonetheless, there are still two other combinations possible, resulting in a combined match count of $3 \cdot 2 = 6$, as opposed to $3 \cdot 1 = 3$, if the sum of constraints would have been used to determine the constraint factor $f(X_i)$.
The described procedure for determining match counts of disjunct patterns is thoroughly tested and works reliably for patterns that have a similar structure, compared to those presented in the example figures. Since this procedure is not formally proven, it can not be guaranteed that it will work for every conceivable type of pattern. However, looking at biochemistry, most patterns of this problem domain are fairly similar to those presented here and rarely have the need for injectivity constraints. Considering this, calculating

match counts in the presented fashion is appropriate for the purpose of simulating biochemical processes.
Using equation 17 and a set of constraint factors, the correct match count for disjunct patterns can be determined through the match count of its constituent sub-patterns. With the help of that match count, the activity of corresponding rules can be determined. In order to apply the rule, not only is the match count to its LHS is needed, but at least one match as well, onto which the rule can be applied. In this case, such a match does not exist yet, because only matches to sub-patterns have been found. Therefore, a hybrid match has to be constructed from matches of sub-patterns.

Algorithm 2 Constructing Hybrid Matches from Matches to Sub-Patterns

```
1: function CREATEHYBRIDMATCHSET(mapOfMatchSets)
2:     setOfMatches ← $\emptyset$
3:     for (subPattern, matchSet) $\in$ mapOfMatchSets do
4:         for match $\in$ matchSet do
5:             if $isNotInSet$(setOfMatches, match) then
6:                 $addToSet$(setOfMatches, match)
7:                 $break$
8:     return setOfMatches
```

Algorithm 2 shows how the set of matches, contained by a hybrid match, is constructed in this thesis. The algorithm receives a map of match sets, as parameter. These sets are mapped to their corresponding sub-patterns. The outer loop, in algorithm 2, goes through each sub-pattern match set tuple in the map. The inner loop checks for each match of the given match set, if it is not yet present within the set of matches, returned by this function. If this is the case, the current match is added and the inner loop is exited. Once the inner loop has finished, the outer loop inspects the next sub-pattern match set tuple. This continues until each sub-pattern has a corresponding match in the returned set of matches. A hybrid match, containing this set along with a mapping from original pattern parameters to the corresponding parameters in the constituent sub-patterns, can then be used as a target for rule application. From the graph transformation's point of view, such a hybrid match is virtually identical to an ordinary match.

This preprocessing step should not only reduce the amount of runtime required for matching, but also decrease the storage consumption of Rete-Networks, due to the reduction of the unreasonable sized match count of disjunct patterns. The effects of this hybrid approach are presented and evaluated in the following chapter.

4 Results and Evaluation

The first part of this section presents simulation results produced by the framework and shows that these are plausible. This evaluation is based upon the simulation of two typical biochemical processes, which are introduced and analyzed in subsection 4.1. In the subsequent subsection 4.2, runtimes and memory consumption are investigated in different scenarios and hybrid pattern matching is compared to non-hybrid pattern matching. In addition, it will be evaluated, how each of the used pattern matching tools cope with disjunct patterns, which are appearing frequently in the biochemistry problem domain. In the last subsection 4.3, the performance of the simulation framework presented this thesis is analyzed, using a more complex example from active research. Therefore, the runtimes, using two different pattern matching tools as well as the hybrid and the non-hybrid approach, are compared. In addition, these runtime results are compared with those produced by KaSim (see subsection 5.4), to get an assessment of the performance a state-of-the-art simulation tool can achieve, in this particular problem domain.

4.1 Simulation Results

In this subsection, two different models of biochemical processes are simulated and the obtained results are presented. It will be analyzed, if the results correspond with the expectations that can be drawn a priori from those models. On the other hand, results from other frameworks, which also simulated this model, are used as comparison. The aim is to evaluate, if the simulation works according to the expectations inferred from a model and if it delivers plausible results that are consistent with results of other established simulation tools.

4.1.1 Simulation of the Goldbeter–Koshland Loop

The first biochemical process, modeled and simulated, is the **G**oldbeter–**K**oshland (*GK*) loop [GK81], which was also used by Danos et al. [DFF$^+$07] to demonstrate their KaSim simulation tool. It represents a recurring mechanism that can be found in numerous biochemical signaling pathways. The reason for using this process as an example is that it is very compact and only posses few agent types and rules. Therefore, it is well suited to introduce the style in which simulation results are presented.

The GK loop consists of only 3 agent types: a kinase `K`, a target molecule `T` and a phosphatase `P`. A kinase can bind to a target and through that phosphorylate a site of the target. Conversely, a phosphatase can bind to an already phosphorylated site of `T`, if the site is free. A site bound to a phosphatase is then unphosphorylated again. `P` and `K` can spontaneously detach themselves from `T` at any time. The exact herein used model can be found in the Appendix A.

Figure 4.1 shows a simulation run of the described model. During this run all targets `T` were observed, where both of their sites are in the phosphorylated state, regardless of the binding state. Additionally, all `T`s were observed, whose sites are phosphorylated and unbound. The number of matches on the former pattern is represented by the red plot in figure 4.1 and the latter pattern by the blue plot. According to the model, it is to be expected that both the phosphatases and the kinases will initially bind to free targets.

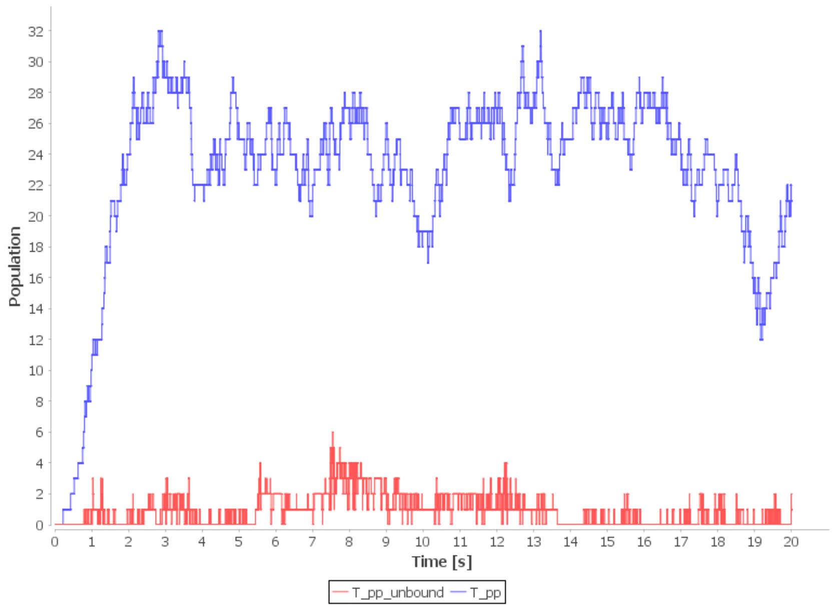


Figure 4.1: Simulation Results – Goldbeter–Koshland Loop (Thesis Framework)

In the case of a kinase binding to a unphosphorylated `T`, it is phosphorylated. Whereas, the binding of a phosphatase to the same `T` would not change the unphosphorylated state. Initially, all agents are unbound and their sites are unphosphorylated. Therefore,

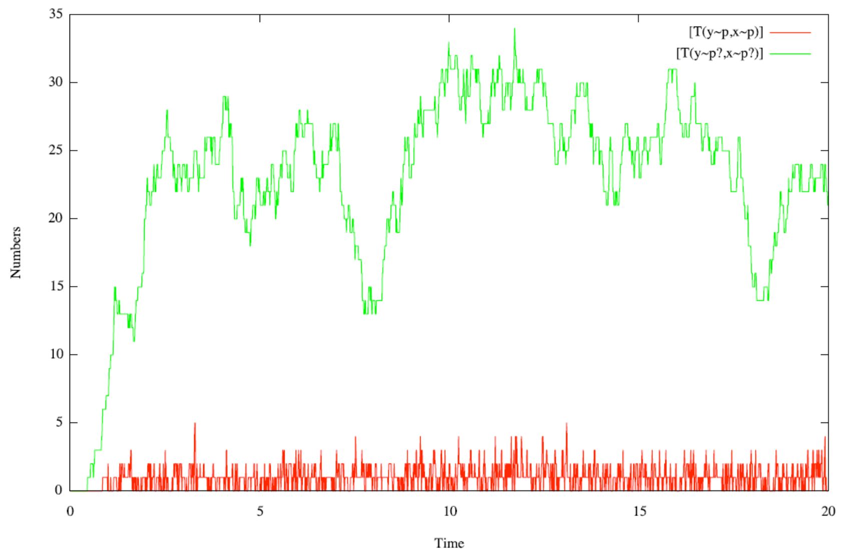


Figure 4.2: Simulation Results – Goldbeter–Koshland Loop (KaSim) [DFF+07]

due to the initially large number of unbound agents, it will lead to a rapid increase in double phosphorylated targets. The red plot in figure 4.1 confirms this expectation. Since the phosphatases reverse phosphorylated states of the targets, a drop of doubly phosphorylated `Ts` has to occur eventually, which can also be seen in the figure. As we can see, this process then begins to oscillate. The blue plot is oscillating at a lower level, since it represents the population of unbound doubly phosphorylated targets. These rarely

occur, because phosphatases as well as kinases simultaneously compete for free sites on targets, which are therefore rarely unoccupied.
Looking at the second figure 4.2, extracted from a paper by Danos et al. [DFF+07] and created with KaSim, a very similar behavior can be observed. A similarly rapid increase of the doubly phosphorylated targets, followed by a subsequent transient oscillation and a low level of unbound targets. However, an exact match of the plots cannot be expected, since both KaSim and this framework are based on a stochastic simulation approach and results are, therefore, subject to statistical fluctuations. Nonetheless, both simulations show a similar activity within 20 seconds of simulation time and have quite similar average values of their respective pattern populations.

4.1.2 Simulation of EGF Signal Pathway

The EGF signaling pathway is another example of a process that can be simulated with the framework of this thesis. As explained in subsection 2.1.1, this is a regulatory mechanism for the growth of skin cells. Since this process plays a role in many control mechanisms of the human body, it is still the subject of active research and a popular example to demonstrate simulation frameworks. Danos et al. [DFF+07] also used the EGF pathway to present KaSim simulation results of Kappa models. Therefore, the EGF signal pathway will also serve as a demonstrator for the framework presented in this thesis. The full model can be found in Appendix B.

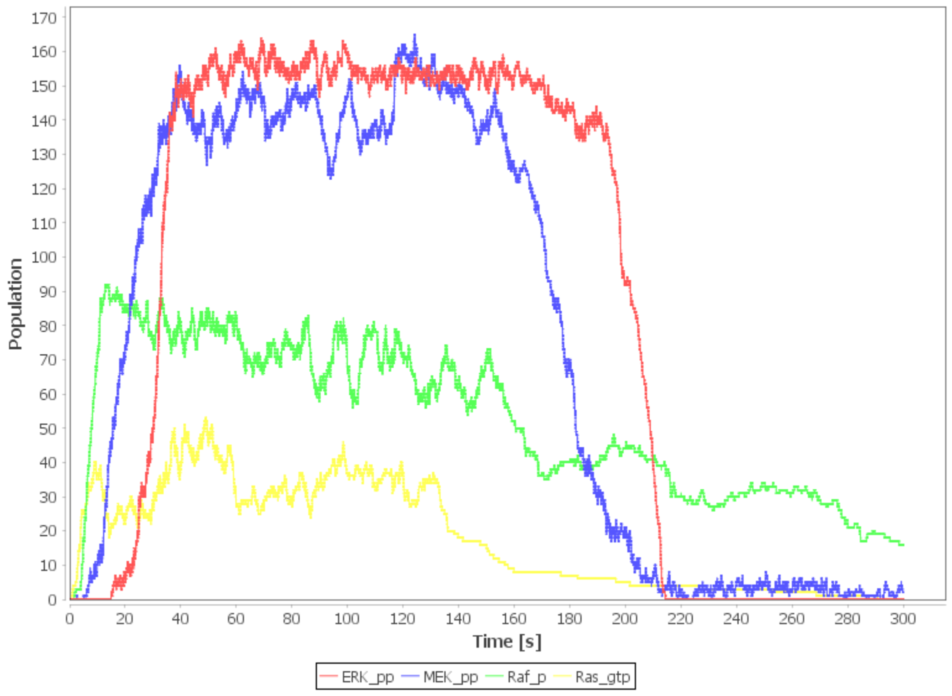


Figure 4.3: Simulation Results – EGF (Thesis Framework)

The process itself is explained in detail in subsection 2.1.1 and basically represents a self-regulating cascade of molecular reactions. During the simulation, population counts of Ras, Raf, MEK and ERK molecules were observed. These molecules are especially

interesting because they are important for propagating the external signal, triggered by an EGF, to the nucleus. Consequently, the ERK molecule represents the end of the signal chain and in reality would then cause the cell nucleus to initiate growth processes.
The process begins by bonding EGFs to EGFRs, whereby the EGFRs activate adjacent EGFRs as a result. After several intermediate steps, the site states of Ras molecules are changed to the GTP state, becoming RasGTP molecules. Note that through EGFRs activating their neighboring EGFRs, a single EGF may cause multiple Ras molecules to become RasGTP molecules. As shown in figure 4.3, this causes a rapid increase in the RasGTP population (yellow plot). Through the bonding of RasGTP molecules with Raf molecules, an equally strong increase of Raf molecules with phosphorylated sites is caused (green plot). Since Raf molecules phosphorylate by binding MEK molecules, their population increases as well (blue plot). Analogously, ERK molecules are phosphorylated by MEK molecules, which, therefore, results in a similarly strong increase in their population (red plot). According to theory, this cascade should begin to stagnate, since all mentioned molecules can spontaneously dephosphorylate. Additionally, RasGTP molecules can form bonds with each other. This self-regulation can be observed very well in the simulation. First, the rise in all molecular populations begins to stagnate. After that, a phase of oscillation and slight decline follows. Finally, the successive decrease of all curves, starting with the yellow plot for RasGTP, can be observed. This continues until the complete flattening of the red plot, representing phosphorylated ERK molecules.

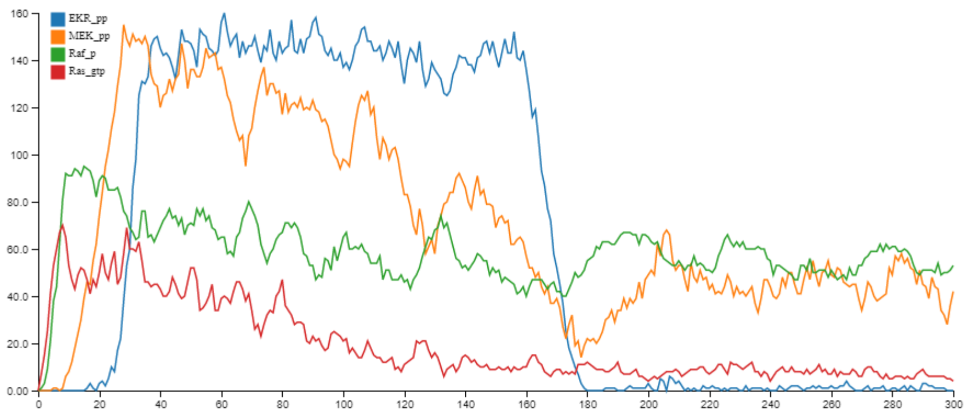


Figure 4.4: Simulation Results – EGF (KaSim)

Using the same model for a simulation in KaSim, a similar behavior can be observed in figure 4.4. In this graph, RasGTP is represented by a red plot, Raf by a green plot, MEK by a yellow plot and ERK by a blue plot. When we compare the plots in both figures 4.3 and 4.4 with each other, similar population changes over time as well as similar peak values can be observed. When considering the fact that these are stochastic simulations, we can assume that both simulation frameworks essentially deliver the same results.

4.2 Evaluation of Hybrid Pattern Matching

Subsection 3.4.3 introduced disjunct patterns, a class of patterns that can cause negative runtime behavior and high memory consumption, when used in general-purpose pattern matching tools. Consequently, this problem also afflicts the simulation framework of this thesis, which makes use of these tools. In order to avoid problems caused by these patterns, hybrid pattern matching was introduced in subsection 4.2. For this reason, the following subsections will analyze the effects of disjunct patterns on simulation runtime and memory consumption. The main focus rests on investigating to what extent the hybrid pattern matching approach can reduce the negative performance effects of disjunct patterns. In subsection 4.2.1, simulation runtime and memory consumption is measured, while varying the number of simulation entities. Subsection 4.2.2, on the the other hand, measures the same metrics, while varying the number of parameters in patterns.

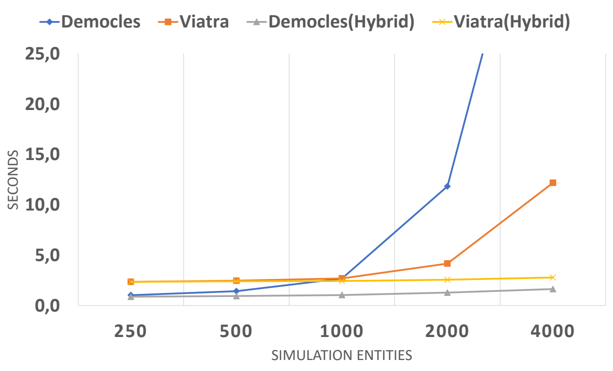
4.2.1 Effects of Model Size variation

This subsection examines how runtime and memory consumption behave with different model sizes for the same pattern. Here, model size implies the number of instanced agents of a certain type. For this purpose, the model in Listing 4.1 is used, which has two agent types `A` and `B` and a rule with a disjunct pattern in its LHS. Thus, two unconnected agents `A` and `B` are to be found as preconditions of the rule. 125 agents of type `A` and 125 agents of type `B` are created as initial conditions. These initial values are then doubled for each run, up to a number of 2000 instances each. For each run, the runtime and memory consumption during 300 iterations is measured. Due to the properties of disjunct patterns, up to $125 \cdot 125 = 15625$ matches can be found in the first run and up to $2000 \cdot 2000 = 4 \cdot 10^6$ matches can be found in the last run. The rule `r1` is designed in such a way, that the simulation activity remains at a constant high level during a single run and at about the same level for each run, independent of model size. This was done to prevent a simulation finishing to fast, due to its activity reaching zero early and resulting in inactivity during each subsequent step. Since inactivity takes almost no computational effort, simulations reaching zero activity by chance would create outliers in the set of measurements.

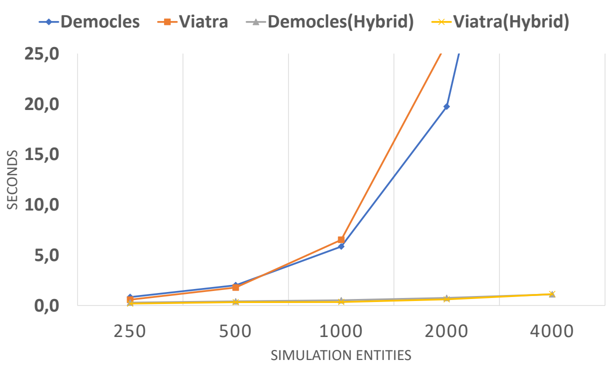
Listing 4.1: Variation of Model Size – Evaluation Model

```
agent A(x, y)
agent B(x, y)

init i1 125 {A(x[free]), B(x[free])}

rule r1 {A(x[free]), B(x[free])} <-> {A(x[1]), B(x[1])} @ [1, 350]

terminate it iterations=300
```

Figure 4.5(a) shows runtime results of the setup phase. Figure 4.5(b) shows simulation runtimes. The series of measurements were made using the Viatra tool and the Democles tool, each with and without using the hybrid pattern matching approach. The number of model entities is given on the x-axis, while the runtime in seconds is given on the y-axis. When looking at the runtimes of the setup phase in figure 4.5(a), we can see that there is a quasi-exponential slope in the measurement series for runs without using the hybrid approach (orange and blue plots). However, runtime measurements of simulation runs with the hybrid approach activated (yellow and gray plots), show only a linear growth,



(a) **Setup Runtimes**



(b) **Simulation Runtimes**

Figure 4.5: Variation of Model Size – Runtime Measurements

under variation of the model size. The offset of the yellow and orange plots in figure 4.5(a) and figure 4.6, representing the measurement series using the Viatra tool, comes from the necessary conversion into the Viatra DSL. Code generation, parsing and interpretation through the Viatra framework always requires some additional time (∼1.5 seconds), as opposed to the Democles tool. It is also interesting to note that Viatra seems to perform some optimizations in the background, which cause the exponential runtime growth (orange plot) to be slightly lower than that of Democles (blue plot).

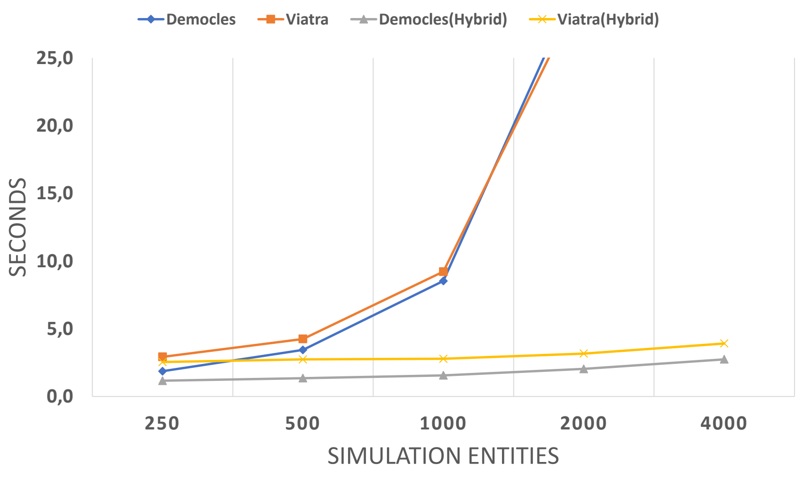


Figure 4.6: Variation of Model Size – Total Runtime

Simulation runtimes after the setup phase behave very similar, while increasing the model size. In figure 4.5(b), the blue and orange plots that represent runtimes using Viatra and Democles without a hybrid approach, also describe exponential slopes. However, runs under the use of hybrid pattern matching only experience a linear increase in runtimes. Viatra (yellow) and Democles (gray) are almost equally fast, with a slight advantage for Democles. Total runtimes, i.e., the sum of setup phase and simulation, plotted in figure 4.6, unsurprisingly show the same behavior: Exponential increases without the hybrid approach (orange and blue plots), linear increases with hybrid pattern matching and a slight offset using Viatra (yellow plot).

These exponential increases in runtime were to be expected, since with each run, through doubling the number of agents the number of matches quadrupled. Conversely, the hybrid approach showed linear behavior, because only matches to sub-patterns had to be found, which only doubled in each run. When looking at the memory consumption in figure 4.7,

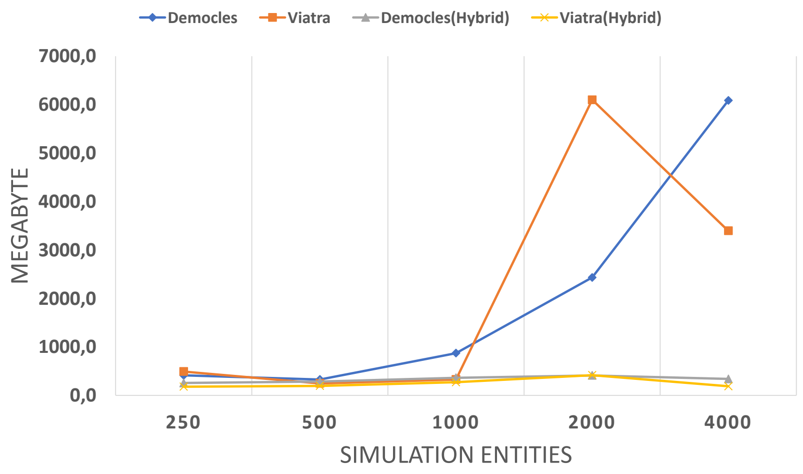


Figure 4.7: Variation of Model Size – Memory Usage

this reasoning is directly reflected. Instead of the runtime, used heap memory is given in megabytes on the y-axis. The blue plot, representing Democles without the hybrid approach grows exponentially. The reason for this is the fact that the Rete-Network has to store the exponentially growing number of matches. Hence, the memory consumption increases. Looking at Viatra, the plot in orange basically shows the same behavior. The sudden sharp bend in the plot is most likely the result of an internal optimization. With the hybrid approach, the Rete-Network does not have to store an exponentially growing match set, only a linearly growing match set. Therefore, the storage requirement only grows in a linear fashion, when using the hybrid approach.

4.2.2 Effects of Pattern Size variation

This subsection examines how runtime and memory consumption behave, while keeping model size constant and varying pattern size. For this purpose, 5 different models are used, the first model is shown in listing 4.2 and is quite similar to the model in the previous subsection. The differences are: a smaller initial amount of agent instances, only 8 each, and a limit of 100 iterations.

Listing 4.2: Variation of Pattern Size – Evaluation Model 1

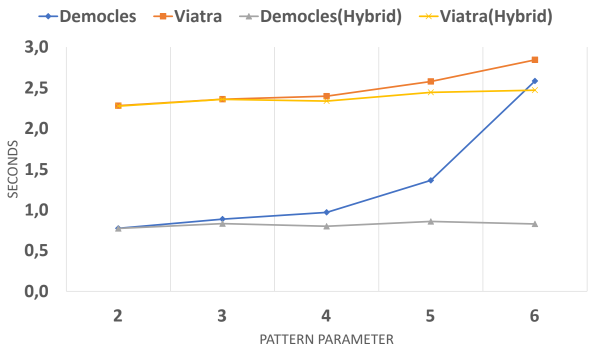
```
agent A(x, y)
agent B(x, y)

init i1 8 {A(x[free]), B(x[free])}

rule r1 {A(x[free]), B(x[free])} <-> {A(x[1]), B(x[1])} @ [1, 200]

terminate it iterations=100
```

As in the previous subsection, model size refers to the number of instanced agents of a certain type. Pattern size describes the number of parameters in a pattern signature. Each subsequent model has one additional parameter in the rule's patterns, up to 6, as can be seen in listing 4.3. In every model, all agent types are initialized with the same number of 8 instances. All LHS rule patterns are disjunct, which produces up to $8 \cdot 8 = 64$ matches, in case of Model 1, and up to $8^6 = 262144$ matches, in the case of Model 5.

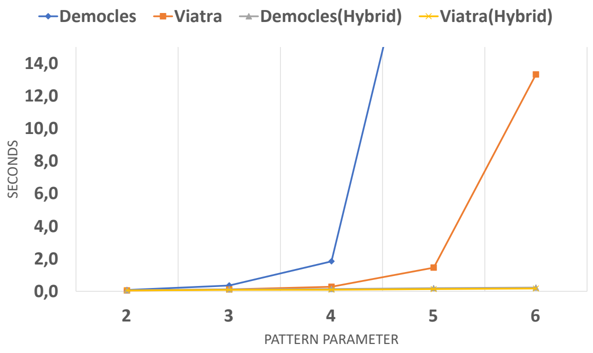
Listing 4.3: Variation of Pattern Size – Evaluation Model 5

```
agent A(x, y)
agent B(x, y)
agent C(x, y)
agent D(x, y)
agent F(x, y)
agent G(x, y)

init i1 8 {A(x[free]), B(x[free]), C(x[free]), D(x[free]), F(x[free]),
G(x[free])}

rule r5 {A(x[free]), B(x[free]), C(x[free]), D(x[free]), F(x[free]),
G(x[free])} <-> {A(x[1]), B(x[1]), C(x[2]), D(x[2]), F(x[3]), G(x[3])}
@ [1, 600000]

terminate it iterations=100
```

Similar to the previous subsection, rules `r1` to `r5` are designed in such a way, that the simulation activity remains at a constant high level during a single run and at about the same level for each run, independent of model size. As in subsection 4.2.1, this was done to make individual runs comparable and prevent outliers.



(a) **Setup Runtimes**



(b) **Simulation Runtimes**

Figure 4.8: Variation of Pattern Size – Runtime Measurements

Figure 4.8(a) shows runtime results of the setup phase, whereas figure 4.8(b) shows simulation runtime measurements. Similar to the previous subsection, the series of measurements were made using the Viatra tool and the Democles tool, each with and without using the hybrid pattern matching approach. The number of pattern parameters is given on the x-axis, while the runtime in seconds is given on the y-axis.
When looking at the runtimes of the setup phase in figure 4.8(a), we can see a simi-

lar behavior as in the previous subsection. Again, we can observe that there is a quasi exponential slope in the measurement series representing runs without using the hybrid approach (orange and blue plots). Furthermore, the runtime measurements of simulation runs with the hybrid approach activated (yellow and gray), show a linear growth as well. The offset of the yellow and orange plots in figure 4.8(a) and figure 4.9, representing measurements using the Viatra tool, are caused by the necessary conversion steps into the Viatra DSL. However, Democles without a hybrid approach (blue plot) reacts with a steeper runtime increase than Viatra (orange plot). This could as well be a result of some internal optimization in the Viatra framework.

In subsection 4.2.1, where the model size was varied, simulation runtimes after the setup phase behave very similar to those in this subsection, where the pattern size was increased. In figure 4.8(b) the blue and orange plots, representing the runtime, while using Viatra and Democles without the hybrid approach, also describe exponential slopes. Furthermore, Democles seems to react worse than Viatra to the increase in pattern size. Runtimes using the hybrid approach describe a shallow linear increase. Simulations using Viatra (yellow plot) or Democles (gray plot) are again almost equally fast, with a slight advantage for Viatra.

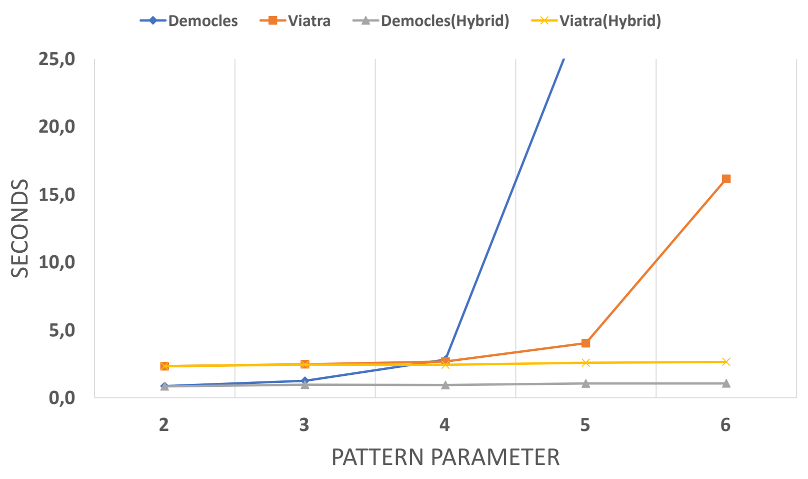


Figure 4.9: Variation of Pattern Size – Total Runtime

Since total runtime measurements in figure 4.9 represent the sums of setup and simulation runtimes, they show the same behavior as previously discussed measurements. Again, with a slight offset added to simulation runs that were using Viatra (yellow and orange plot).

As in the previous chapter, these exponential runtime growths were to be expected here as well. In this case, by adding a parameter each run, the number of matches increased eightfold each time. With the hybrid approach, the number of matches only increased by 8, with each additional parameter, because matches for the sub-patterns could be searched for independently of each other. Consequently, this results in an almost constant, at least a very slight linear increase in runtime. When looking at the memory consumption in figure 4.10, this reasoning is directly reflected here as well. Similar to the previous subsection, used heap memory is given in megabytes on the y-axis. The

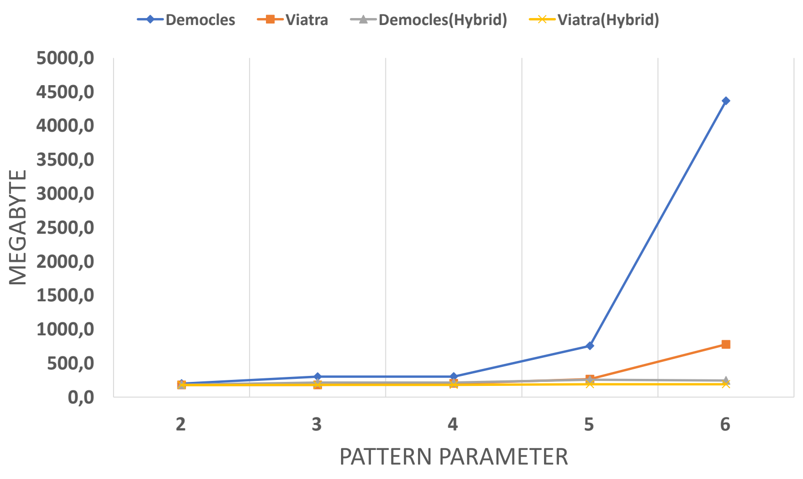


Figure 4.10: Variation of Pattern Size – Memory Usage

blue plot, representing simulation runs using Democles without hybrid approach, grows exponentially. The observed behavior is, again, caused by the fact that the Rete-Network has to store the exponentially growing number of matches. However, a different behavior can be observed when using the hybrid approach, where the number of matches for each partial pattern does not increase. Instead, the same amount of matches is added for each additional sub-pattern, leading to a linear increase in memory requirements. In fact, the actual memory consumption for simulation runs using the hybrid approach (gray and yellow plots), behaves even better than expected. In figure 4.10, said plots indicate that memory consumption remains virtually constant, which might be caused by internal optimizations in the used pattern matching tools.

4.3 Runtime comparison with KaSim

In this subsection, a series of runtime measurements are performed, using the EGF signal pathway model from appendix B. The aim is to investigate, how much performance gain can actually be achieved using the hybrid pattern matching approach in a realistic scenario. In addition, a series of measurements is created using KaSim, which simulates a Kappa model of the EGF signal pathway. The results will be compared to the series of measurements that were gained using the framework presented in this thesis. This serves the purpose of getting an indicator, how the performance of this simulation framework compares to a state-of-the-art domain-specific simulation tool.
For this purpose, besides KaSim, runtime measurements were taken for EGF simulation runs using Democles and Viatra, with and without the hybrid approach. For each run, the model size was varied, meaning, that the initial conditions, as listed in appendix B, were first scaled to 25%, then to 50%, to 75% and finally, up to 125%.
The results of these runtime measurements are shown in figure 4.11, where the relative model size is given on the x-axis. The runtime, in seconds, is given on the y-axis, which is scaled logarithmically. The scaling is necessary because otherwise no reasonable discussion of the results would be possible, due to the large differences in runtime.
When looking at the measurements created using Democles, we can see that for each

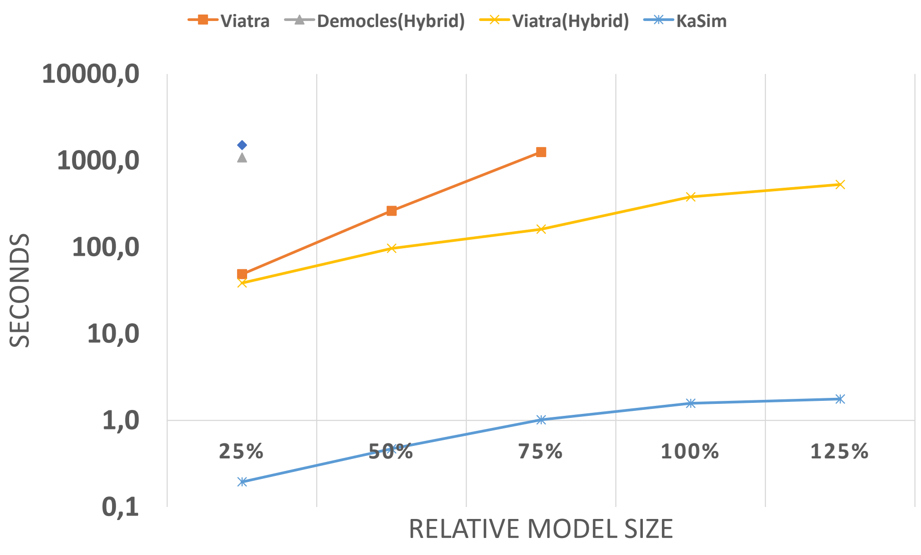


Figure 4.11: EGF Runtime Measurements

series, only a single measuring point is plotted (gray and dark blue data points). The reason for this are the large runtimes that already occurred at 25% model size. These runtimes would have multiplied again at 50% model size. At this size, each simulation run was aborted after the runtime exceeded three times the runtime of a model at 25% size. We can see that the hybrid approach (gray data point) is a bit faster than the conventional approach using just Democles. However, if one compares this runtime with the other measurement series, it becomes clear that there is no point in continuing to make measurements using Democles with bigger model sizes. As it turns out, Democles does not seem to handle larger numbers of patterns (∼120 in case of EGF) as well, as Viatra. Additionally, when we look at some patterns of the EGF model, we can find some examples that are quite large, i.e., they have many signature nodes which define various link states and or site states. As indicated in the previous subsection, Democles does not react as well to patterns with this many parameters, as Viatra. The series of measurements created by simulation runs using Viatra without hybrid approach (orange plot), supports the impression that Viatra is able to handle this kind of model a bit better than Democles. This linear slope of the orange plot on a logarithmic scale, implies an exponential increase of the runtime in a linear scale. Nonetheless, it does start at a lower level than any measurement series created by using Democles. This performance advantage could possibly be the result of internal optimizations in the Viatra framework as well.
The measurement series created by simulation runs using Viatra with the hybrid approach (yellow plot) starts at a slightly lower level, compared to the previously discussed results and continues with an approximately logarithmic slope. This slope corresponds to a linear increase in runtime, when using a non-logarithmic scale. Although this is not like the extreme increase in performance, as observed in subsections 4.2.1 and 4.2.2, when using synthetic examples, it nevertheless represents a better runtime behavior than conventional pattern matching approaches. About half of the patterns in the EGF model are disjunct and would lead to an exponential runtime increase, when using just Democles or

Viatra. This problem was avoided with the hybrid approach, resulting in the framework achieving a linear growth of the runtime, with linear growth of the model. Keeping this in mind, the comparison with the measurement series created by KaSim (light blue plot), is not as bad as it seems on the first glance. Despite being slower by a quasi constant factor of about 200, the hybrid approach managed to perform with a linear increase in runtime, comparable to the KaSim tool. As expected, the domain-specific tool is massively faster because it is highly optimized. This level of performance is probably not achievable using a general-purpose pattern matching tool.

5 Related Works

Rule-based simulation of biochemical processes has been a research topic for quite some time and, therefore, spawned a variety of different approaches to the modeling and simulation of said processes. For this reason, some of the more widely used modeling and simulation tools are presented in this section.

5.1 BioNetGen

Bio**N**et**G**en (*BNG*) is a rule-based modeling framework for complex biochemical systems, initially developed by Blinov et al. [BFGH04]. The BNG framework, in its current state [HHT+16], consists of a modeling language and a collection of open-source simulation tools. Like Kappa, the **B**io**N**et**G**en **L**anguage (*BNGL*) enables a text based concise description of large reaction networks.
Models expressed in the BNGL consist of two main components: Definitions of biomolecules and rules that define possible interreactions between biomolecules. The process of defining rules in BNGL is largely the same as in Kappa, where rules can be atomic and fall in one of 5 rule categories (binding, unbinding, modification, creation and deletion) or rules are a mixture of atomic rules (see subsection 2.2.2). The definition of agents is slightly different on a syntactical level, but roughly the same on a conceptual level.

Listing 5.1: BNGL – Example Block

```
begin molecule types
        A(x , y)
        B(z ~p ~q)
end molecule types
```

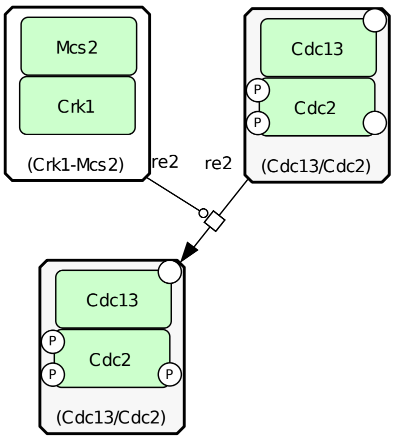
Instead of using keywords, e.g., `agent`, the user defines blocks in BNGL, as shown in listing 5.1, in which all rules, agents and observables must be described. Each block is started with a `begin` statement, followed by the block type, e.g., `molecule types`, after which all definitions for instances of the given type follow. Every block is closed with an `end` statement and its type.
The interesting aspect of the BNG framework is the fact that a model, created in the BNGL, can be simulated using different methods. The simulation method type is a parameter defined at the end of a BNGL model, which leads to the automatic selection of the required tool from the BNG framework. A rule-based model, created in BNGL, can be simulated using a stochastic method, which is based on Gillespie's algorithm (see subsection 2.2.1), therefore, being comparable to the technique presented in this thesis. On the other hand, BNG provides a method to translate a rule-based model to a system of ODEs. Hence, it is possible to simulate rule-based models using the traditional approach, assuming biochemical reaction systems to be continuous and deterministic. The latter is still a viable method for simulating smaller models. Once the modeled biochemical system reaches a certain size, the creators of BNG recommend using simulation tools based on stochastic methods.

5.2 RuleMonkey

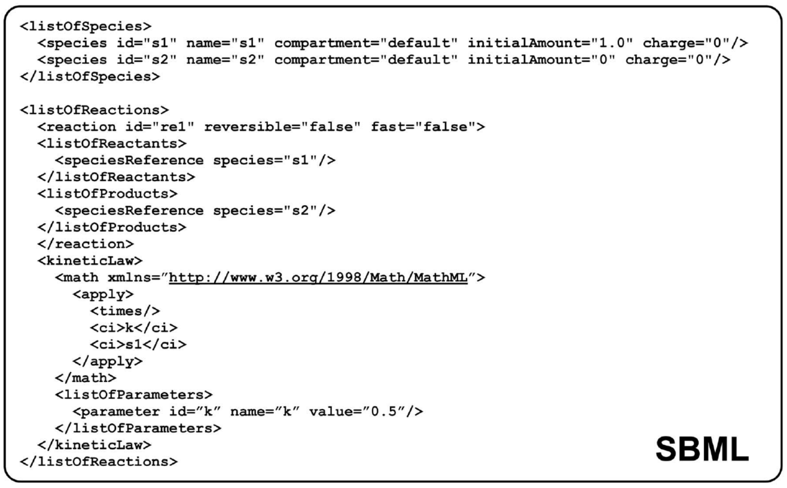
RuleMonkey, developed by Colvin et al. [CMG+10], is a tool that enables the simulation of rule-based models, using a method similar to Gillespie's algorithm. In contrast to Kappa or BioNetGen, RuleMonkey does not have its own DSL that could be used to model biochemical systems. Instead, it is compliant to rule-based models encoded in BNGL. Hence, any model that is created using BNGL can be simulated with RuleMonkey. Like the algorithm used in this framework, the stochastic simulation in RuleMonkey is a network-free approach. That means a generation of the complete reaction network is not required, in contrast to some simulation approaches implemented in BioNetGen. A non network-free approach needs to enumerate all possible species that can be derived from a molecule with multiple states. Using this and the set of rules, a network is generated, which is used to track instances of these molecular species in the simulation. The advantage of network-free approaches is that they don't have an initial generation time and they do not need to keep the complete reaction network in memory. The downside is the fact that simulation steps are slower, because matches to rule preconditions have to be found in some way.

5.3 CellDesigner

Cell**D**esigner (*CD*) is a modeling tool for biochemical networks, developed by Funahashi et al. [FMJ+08]. In contrast to Kappa and BioNetGen, in which models are described textually, CellDesigner supports a graphical notation of biochemical processes using a visual editor. In this case, biochemical networks, such as signal pathways, are represented by process diagrams. This representation lends itself naturally to the semantics of biochemical reaction networks, since a biochemical reaction usually represents a state transition of molecules.
The notation system that is used in the process diagrams is called **S**ystems **B**iology **G**raphical **N**otation (*SBGN*), developed by an international community. SBGN allows the representation of diverse biological objects as well as interactions and is designed to be semantically and visually unambiguous. Within the SBGN notation, each node represents the state of molecules and each arrow represents state transitions, among the states of a molecule (see figure 5.1 (a)).
CellDesigner models are encoded in the **S**ystems **B**iology **M**arkup **L**anguage (*SBML*), which is a tool-neutral format based on the XML standard for representing models of biochemical reaction networks. In SBML, model properties are stored under certain tags, for example, molecular species, such as protein types, are stored under the <listOfSpecies> tag (see figure 5.1 (b)). The idea behind the introduction of SBML was to create a portable format, capable of representing biochemical reaction network models, which can be used in different software systems. As a consequence, the CellDesigner framework comes with a plethora of different open source simulation tools that all use SBML encoded models.
Like the BioNetGen framework, the CellDesigner software can run ODE-based simulations as well as stochastic simulations. The application itself is coded in Java, which has the advantage of being portable and, thus, able to run on many different platforms.



(a) **SBGN Example**



```
<listOfSpecies>
  <species id="s1" name="s1" compartment="default" initialAmount="1.0" charge="0"/>
  <species id="s2" name="s2" compartment="default" initialAmount="0" charge="0"/>
</listOfSpecies>

<listOfReactions>
  <reaction id="re1" reversible="false" fast="false">
  <listOfReactants>
    <speciesReference species="s1"/>
  </listOfReactants>
  <listOfProducts>
    <speciesReference species="s2"/>
  </listOfProducts>
  </reaction>
  <kineticLaw>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply>
        <times/>
        <ci>k</ci>
        <ci>s1</ci>
      </apply>
    </math>
    <listOfParameters>
      <parameter id="k" name="k" value="0.5"/>
    </listOfParameters>
  </kineticLaw>
</listOfReactions>
```

(b) **SBML Example**

Figure 5.1: CellDesigner – SBGN and SBML [FMJ+08]

Additionally, CD comes with a database of existing models, which can be imported and used in simulations.

5.4 KaSim

The Kappa creators developed their own rule-based simulation tool, called KaSim [BFKF18]. Like most rule-based methods, for example, BioNetGen's integrated simulation tool, KaSim is based on Gillespies's algorithm (see subsection 2.2.1) and, therefore, performs a stochastic simulation of biochemical processes. KaSim natively supports Kappa models, which contain the specification of the model, i.e., rules, molecular species and initial conditions.
The state of the simulation is defined by the so called *mixture*, which is a collection of instances created from molecular species, defined by the Kappa model. The mixture can be seen as a "soup" of molecules, representing the system at any point in time, and is basically one big graph consisting of molecules, connected through subgraphs. During the simulation, reactions, represented by rule applications, occur with a certain probability. As stated in subsection 2.2.1, in order to calculate said probability the of number matches to a rule's LHS must be determined. In contrast to the framework presented in this thesis, KaSim does not use a general-purpose pattern matching tool to determine the number of matches. Instead of counting subgraphs that satisfy a rule's LHS, KaSim maintains and administers all embeddings from rules into the mixture graph [BFKF18].
The KaSim application has a web front end, which makes it possible to define Kappa models and run simulations from inside a web browser.

6 Conclusion

The overarching goal of this thesis was to create a framework for rule-based simulation of biochemical processes. In this type of simulation, it is mandatory to find the correct candidates for rule applications, which model molecular reactions. For this purpose, an important task was to integrate two different general-purpose pattern matching tools, with the intent to evaluate the performance of the simulation framework, using both tools in different scenarios. The final task was to compare the framework's performance measurements and simulation results to those achieved by the domain-specific simulation tool KaSim.

For this purpose, the simulation framework was designed for modularity, with the aim of being able to switch between different pattern matching tools as well as pattern matching strategies with little effort. First, a new DSL inspired by Kappa was designed, which is used to model biochemical processes. Such a model carries the information about the properties of all simulation entities as well as their initial number and rules, which describe the way entities can react with each other.
The simulation was implemented according to the principle of stochastic simulation for biochemical processes, described by Gillespie's algorithm. In this stochastic simulation the application probability of rules is determined using the match counts to their application candidates. Hence, both integrated pattern matchers play an important role by finding these required matches. However, general-purpose pattern matching tools run into a performance problem, when disjunct patterns are used, due to exponential growth of their match numbers. Since disjunct patterns appear frequently in biochemistry, the hybrid approach to pattern matching was developed in this thesis. The hybrid approach avoids exponential match growth, by splitting the disjunct patterns into their non-disjunct sub-patterns.
Simulation results and their plausibility were discussed using two different simulation models. Additionally, the results were compared with those from KaSim, an established simulation tool that is actively used in research. As a result, in subsection 4.1, we came to the conclusion that the simulation produces plausible results for both simulation models. One reason for this conclusion was that the results coincided with expectations that could be inferred from the models. On the other hand, the results coincided with those from KaSim.
In order to determine how effective the hybrid approach actually is, measurements of runtime and memory consumption during simulation runs were performed, using synthetic models and varying different model parameters. As expected, during this analysis in subsection 4.2, the exponential growth in match numbers, caused by disjunct patterns, inevitably lead to an exponential growth in runtime and memory consumption. It became clear how important it was to develop hybrid pattern matching, which tackles the problem of disjunct patterns, within the context of biochemistry. A remarkable increase in performance could be observed, when using the hybrid approach. Not only did runtimes decrease noticeably overall, but the memory consumption decreased as well. Furthermore, the increase in memory consumption, as well as the runtime increase was limited to a lin-

ear growth.
Finally, simulation runtimes of the EGF signal pathway were measured, which represents a more realistic simulation model and served as an example of a biochemical process from active research. During this, the simulation framework was used with either Viatra or Democles and with activated and deactivated hybrid pattern matching. The runtime results were then compared with those of KaSim. Since the rules in the EGF signal pathway contain some disjunct patterns, exponential runtime increases could again be observed in simulation runs without the hybrid approach. It turned out that Viatra can cope somewhat better with the greater number and the type of patterns found in the EGF model, compared to Democles. The hybrid approach was consistently faster, but in this case could not achieve the remarkable gains it had achieved using the synthetic examples, since the EGF signal pathway does not entirely consist of disjunct patterns. It also contains at least as many non-disjunct patterns, where the hybrid approach can not cause any runtime improvements. The comparison with KaSim demonstrated that hybrid pattern matching is not the silver bullet to removing all performance deficits, which general-purpose pattern matching tools have, compared to domain-specific tools. Such a tool still performs better by a large margin and, thus, delivers a shorter runtime than each of the two pattern matching tools with the activated hybrid approach. However, the upside is that the runtime increase, using the hybrid approach, is not exponential anymore, but linear, which counts as a real success. In addition, general-purpose pattern matching tools still have the advantage of higher expressiveness, which in principle grants the ability to describe much more complex patterns. This allows the simulation framework to be extended through additional features, such as complex constraints for rule applications.

In retrospect, one reason for the performance issues of Democles can certainly be found in the somewhat cumbersome modeling of the Reaction Container metamodel. One of the problems might be the representation of links between sites, which is expressed by using SimLinkState objects. In the future, this should be solved by directly using references between sites. This would simplify patterns passed to the pattern matchers, which would benefit not only Democles but Viatra as well. Additionally, it would not require thousands of objects to be instantiated, just to express links between other objects. Consequently, the costly deletion of these link objects, within the context of EMF, could be avoided as well. In addition to that, using distinct object types for agents, instead of type attributes, could potentially improve the performance of pattern matching, reducing the size of patterns even more.
Furthermore, it would be reasonable to make more use of the expressiveness provided by general-purpose pattern matching tools, by introducing, for example, complex constraints for rule application. A conceivable potential future improvement, taking up on this idea, could be the implementation of a collision detection, which, e.g., detects and prevents collisions between agents that otherwise would have occurred, if the rule would have been applied.
In addition, comparing this framework to other simulation tools, besides KaSim, could potentially be interesting. Especially after implementing additional features, e.g, collision detection, that other tools might not have.

As a final remark, some aspects of the implementation leave room for improvements, with respect to performance. For example, when models in the DSL reach dimensions comparable to those of the EGF model, syntax checking becomes quite slow. This and other minor issues make for some simple improvements, apart from optimizing patterns and metamodels.

In conclusion, we can say that despite some drawbacks revealed in the evaluation, this thesis, nevertheless, produced some major achievements. A framework was implemented that can perform a rule-based simulation of biochemical processes. A DSL has been developed to model biochemical processes. In addition, general-purpose pattern matching tools were used, which offer great potential for extensibility of the framework, due to their expressiveness. Finally, by developing the hybrid pattern matching approach, the problem of exponential runtime increase caused by disjunct patterns was tackled successfully, within the context of biochemistry.

References

[BCP12] Bernardo, Marco ; Cortellessa, Vittorio ; Pierantonio, Alfonso: *Formal Methods for Model-Driven Engineering.* Springer-Verlag Berlin Heidelberg, 2012

[BCW12] Brambilla, Marco ; Cabot, Jordi ; Wimmer, Manuel: *Model-Driven Software Engineering in Practice.* Morgan and Claypool, 2012

[BFGH04] Blinov, Michael L. ; Faeder, James R. ; Goldstein, Byron ; Hlavacek, William S.: BioNetGen: software for rule-based modeling of signal transduction based on the interactions of molecular domains. In: *Bioinformatics* 20 (2004), Nr. 17, S. 3289–3291

[BFKF18] Boutillier, Pierre ; Feret, Jérôme ; Krivine, Jean ; Fontana, Walter ; KappaLanguage.org (Hrsg.): *The Kappa Language and Kappa Tools.* v4. : KappaLanguage.org, 2018. https://kappalanguage.org/sites/kappalanguage.org/files/inline-files/Kappa_Manual_1.pdf

[CC79] Carpenter, Graham ; Cohen, Stanley: Epidermal Growth Factor. In: *Annual Review of Biochemistry* 48 (1979), Nr. 1, S. 193–216

[CHA+18] C.Wolff, Antonio ; Hammond, M. Elizabeth H. ; Allison, Kimberly H. ; Harvey, Brittany E. ; Mangu, Pamela B. ; Bartlett, John M. ; Bilous, Michael ; Ellis, Ian O. ; Fitzgibbons, Patrick ; Hanna, Wedad ; Jenkins, Robert B. ; Press, Michael F. ; Spears, Patricia A. ; Vance, Gail H. ; Viale, Giuseppe ; McShane, Lisa M. ; Dowsett, Mitchell: Human Epidermal Growth Factor Receptor 2 Testing in Breast Cancer: American Society of Clinical Oncology/ College of American Pathologists Clinical Practice Guideline Focused Update. In: *Journal of Clinical Oncology* (2018)

[CMG+10] Colvin, Joshua ; Monine, Michael I. ; Gutenkunst, Ryan N. ; Hlavacek, William S. ; Hoff, Daniel D. ; Posner, Richard G.: RuleMonkey: software for stochastic simulation of rule-based models. In: *BMC Bioinformatics* 11 (2010), Nr. 1, S. 404

[DFF+07] Danos, Vincent ; Feret, Jérôme ; Fontana, Walter ; Harmer, Russell ; Krivine, Jean: Rule-Based Modelling of Cellular Signalling. In: *CONCUR - Concurrency Theory* Bd. 8, Springer Berlin Heidelberg, 2007, S. 17–41

[DFF+09] Danos, Vincent ; Feret, Jérôme ; Fontana, Walter ; Harmer, Russ ; Krivine, Jean: Rule-Based Modelling and Model Perturbation. In: *Transactions on Computational Systems Biology XI.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2009, S. 116–137

[DFFK07] Danos, Vincent ; Feret, Jérôme ; Fontana, Walter ; Krivine, Jean: Scalable Simulation of Cellular Signaling Networks. In: *Programming Languages and Systems*, Springer Berlin Heidelberg, 2007, S. 139–157

[DL04] Danos, Vincent ; Laneve, Cosimo: Formal molecular biology. In: *Theoretical Computer Science* 325 (2004), Nr. 1, S. 69–110. – Computational Systems Biology

[EEPT06] Ehrig, Hartmut ; Ehrig, Karsten ; Prange, Ulrike ; Taentzer, Gabriele: *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag Berlin Heidelberg, 2006

[ELS+10] Engels, Gregor ; Lewerentz, Claus ; Schäfer, Wilhelm ; Schürr, Andy ; Westfechtel, Bernhard: *Graph Transformations and Model-Driven Engineering*. Springer-Verlag Berlin Heidelberg, 2010

[FMJ+08] Funahashi, Akira ; Matsuoka, Yukiko ; Jouraku, Akiya ; Morohashi, Mineo ; Kikuchi, Norihiro ; Kitano, Hiroaki: CellDesigner 3.5: A Versatile Modeling Tool for Biochemical Networks. In: *Proceedings of the IEEE* 96 (2008), Nr. 8, S. 1254–1265

[For82] Forgy, Charles L.: Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. (1982)

[Gil77] Gillespie, Daniel T.: Exact Stochastic Simulation of Coupled Chemical Reactions. In: *The Journal of Physical Chemistry* 81 (1977), Nr. 25, S. 2340–2361

[GK81] Goldbeter, Albert ; Koshland, Daniel E.: An Amplified Sensitivity Arising from Covalent Modification in Biological Systems. In: *Proceedings of the National Academy of Sciences* 78 (1981), Nr. 11, S. 6840–6844

[HHT+16] Harris, Leonard A. ; Hogg, Justin S. ; Tapia, José-Juan ; Sekar, John A. P. ; Gupta, Sanjana ; Korsunsky, Ilya ; Arora, Arshi ; Barua, Dipak ; Sheehan, Robert P. ; Faeder, James R.: BioNetGen 2.2: advances in rule-based modeling. In: *Bioinformatics* 32 (2016), Nr. 21, S. 3366–3368

[KWB01] Kuan, C-T. ; Wikstrand, C. J. ; Bigner, D. D.: EGF mutant receptor vIII as a molecular target in cancer therapy. In: *Endocrine-related cancer* 8 (2001), Nr. 2, S. 83–96

[LBS+04] Lynch, Thomas J. ; Bell, Daphne W. ; Sordella, Raffaella ; Gurubhagavatula, Sarada ; Brannigan, Ross A. Okimotoand Brian W. ; Harris, Patricia L. ; Haserlat, Sara M. ; Supko, Jeffrey G. ; Haluska, Frank G. ; Louis, David N. ; Christiani, David C. ; Settleman, Jeff ; Haber, Daniel A.: Activating Mutations in the Epidermal Growth Factor Receptor Underlying Responsiveness of Non–Small-Cell Lung Cancer to Gefitinib. In: *The new england journal of medicine* 350 (2004), Nr. 21, S. 2129–2139

[LNR+85] Libermann, Towia A. ; Nusbaum, Harris R. ; Razon, Nissim ; Kris, Richard ; Lax, Iritt ; Soreq, Hermona ; Whittle, Nigel ; Waterfield, Michael D. ; Ullrich, Axel ; Schlessinger, Joseph: Amplification and overexpression of the EGF receptor gene in primary human glioblastomas. In: *J Cell Sci* (1985), Nr. 3, S. 161–172

[LV02] Larrosa, Javier ; Valiente, Gabriel: Constraint Satisfaction Algorithms for Graph Pattern Matching. In: *Mathematical. Structures in Comp. Sci.* 12 (2002), Nr. 4, S. 403–422

[MG06] Mens, Tom ; Gorp, Pieter V.: A Taxonomy of Model Transformation. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), S. 125–142. – Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)

[Obj05] Object Management Group (Hrsg.): *Meta Object Facility (MOF) Specification.* v1.4.1. : Object Management Group, July 2005

[OMFK05] Oda, K. ; Matsuoka, Y. ; Funahashi, A. ; Kitano, H.: A comprehensive pathway map of epidermal growth factor receptor signaling. In: *Molecular Systems Biology* 1 (2005), S. 2005.0010

[Sch95] Schürr, Andy: Specification of graph translators with triple graph grammars. In: *Graph-Theoretic Concepts in Computer Science.* Berlin, Heidelberg : Springer Berlin Heidelberg, 1995, S. 151–163

[VBH$^+$16] Varró, Dániel ; Bergmann, Gábor ; Hegedüs Ábel ; Horváth Ákos ; Ráth, István ; Ujhelyi, Zoltán: Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. In: *Software and System Modeling* 15 (2016), Nr. 3, S. 609–629

[VD13] Varró, Gergely ; Deckwerth, Frederik: A Rete Network Construction Algorithm for Incremental Pattern Matching. In: *Theory and Practice of Model Transformations.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, S. 125–140

[VSV05] Varró, Gergely ; Schürr, Andy ; Varró, Dániel: Benchmarking for graph transformation. In: *Proceedings - 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, S. 79–88

[VVS] Varró, Gergely ; Varró, Dániel ; Schürr, Andy: Incremental Graph Pattern Matching. http://www.cs.bme.hu/~gervarro/publication/IncrementalEngine.pdf

[VVS06] Varró, Gergely ; Varró, Dániel ; Schürr, Andy: Incremental Graph Pattern Matching: Data Structures and Initial Experiments. In: *Proceedings of the Second International Workshop on Graph and Model Transformation (GraMoT 2006)*, 2006

[Zü96] Zündorf, Albert: Graph pattern matching in PROGRES. In: *Graph Grammars and Their Application to Computer Science.* Berlin, Heidelberg : Springer Berlin Heidelberg, 1996, S. 454–468

A First Appendix - Model of the Goldbeter–Koshland Loop

The following listing shows the Goldbeter–Koshland loop, modeled as a Reaction Rule DSL model. It was originally created by Danos et al. [DFF+07], using the Kappa language.

Listing A.1: Reaction Rule DSL model of the Goldbeter–Koshland Loop

```
ModelID Goldbeter_Koshland

agent K(a)
agent P(a)
agent T(x{u,p}, y{u,p})

init i1 100 {K(), P(), T(x{u}[free], y{u}[free])}

rule KT_x {K(a[free]), T(x[free])} <-> {K(a[1]), T(x[1])} @ [1, 10]
rule Tp_x {K(a[1]), T(x{u}[1])} -> {K(a[1]), T(x{p}[1])} @ [1]
rule KT_y {K(a[free]), T(y[free])} <-> {K(a[1]), T(y[1])} @ [1, 10]
rule Tp_y {K(a[1]), T(y{u}[1])} -> {K(a[1]), T(y{p}[1])} @ [1]

rule PT_x {P(a[free]), T(x[free])} <-> {P(a[1]), T(x[1])} @ [1, 10]
rule Tu_x {P(a[1]), T(x{p}[1])} -> {P(a[1]), T(x{u}[1])} @ [1]
rule PT_y {P(a[free]), T(y[free])} <-> {P(a[1]), T(y[1])} @ [1, 10]
rule Tu_y {P(a[1]), T(y{p}[1])} -> {P(a[1]), T(y{u}[1])} @ [1]

obs T_pp {T(x{p}[?], y{p}[?])}
obs T_pp_unbound {T(x{p}[free], y{p}[free])}

terminate t1 time=20000
```

B Second Appendix - Model of the EGF Signal Pathway

The following listing shows the EGF signal pathway, modeled as a Reaction Rule DSL model. It was originally created by Danos et al. [DFF+07], using the Kappa language. As a remark, like in the original model, all rule activities are set to 1.

Listing B.1: Reaction Rule DSL model of the EGF Signal Pathway

```
ModelID EGF_Pathway

agent EGF(r{ext, int})
agent EGFR(L{ext, int}, CR, Y992{u,p}, Y1068{u,p}, Y1148{u,p})
agent RasGap(SH2, s)
agent Grb2(SH2, SH3)
agent SoS(a, b, SS{u,p})
agent Shc(PTB, Y318{u,p})
agent Ras(S1S2{gdp, gtp})
agent Raf(x{u,p})
agent PP2A1(s)
agent MEK(s, S218{u,p}, S222{u,p})
agent PP2A2(s)
agent ERK(s, T185{u,p}, Y187{u,p})
agent MKP3(s)

// Activating receptor dimers
// #1 external dimers:
rule EGF_EGFR {EGF(r{ext}[free]), EGFR(L{ext}[free], CR[free])} <->
{EGF(r{ext}[1]), EGFR(L{ext}[1], CR[free])} @ [1,1]

rule EGFR_EGFR {EGFR(L{ext}[bound], CR[free]), EGFR(L{ext}[bound],
CR[free])} <-> {EGFR(L{ext}[bound], CR[1]), EGFR(L{ext}[bound],
CR[1])} @ [1,1]

// #2 simplified phosphorylation (internal or external)
rule EGFR_at_992 {EGFR(CR[bound], Y992{u}[free])} ->
{EGFR(CR[bound], Y992{p}[free])} @ [1]

rule EGFR_at_1068 {EGFR(CR[bound], Y1068{u}[free])} ->
{EGFR(CR[bound], Y1068{p}[free])} @ [1]

rule EGFR_at_1148 {EGFR(CR[bound], Y1148{u}[free])} ->
{EGFR(CR[bound], Y1148{p}[free])} @ [1]

// #3 simplified dephosphorylation (internal or external)
rule _992_op {EGFR(Y992{p}[free])} -> {EGFR(Y992{u}[free])} @ [1]
rule _1068_op {EGFR(Y1068{p}[free])} -> {EGFR(Y1068{u}[free])} @ [1]
rule _1148_op {EGFR(Y1148{p}[free])} -> {EGFR(Y1148{u}[free])} @ [1]

// Internalization, degradation and recycling
// #internalization:
rule int_monomer {EGF(r{ext}[1]), EGFR(L{ext}[1], CR[free])} ->
{EGF(r{int}[1]), EGFR(L{int}[1], CR[free])} @ [0.02]

rule int_dimer {EGF(r{ext}[1]), EGFR(L{ext}[1], CR[2]),
EGF(r{ext}[3]), EGFR(L{ext}[3], CR[2])} -> {EGF(r{int}[1]),
EGFR(L{int}[1], CR[2]), EGF(r{int}[3]), EGFR(L{int}[3], CR[2])} @ [0.02]

// #dissociation:
rule EGFR_EGFR_op {EGFR(L{int}[bound], CR[1]), EGFR(L{int}[bound],
CR[1])} -> {EGFR(L{int}[bound], CR[free]), EGFR(L{int}[bound],
CR[free])} @ [1]

rule EGF_EGFR_op {EGF(r{int}[1]), EGFR(L{int}[1], CR[free])} ->
{EGF(r{int}[free]), EGFR(L{int}[free], CR[free])} @ [1]

// #degradation:
rule deg_EGF {EGF(r{int}[free])} -> {void} @ [1]
rule deg_EGFR {EGFR(L{int}[free], CR[free])} -> {void} @ [1]

// #recycling:
rule rec_EGFR {EGFR(L{int}[free], Y992{u}[free], Y1068{u}[free],
Y1148{u}[free])} -> {EGFR(L{ext}[free], Y992{u}[free], Y1068{u}[free],
Y1148{u}[free])} @ [1]

// SoS and RasGAP recruitment
```

```
rule EGFR_RasGAP {EGFR(Y992{p}[free]), RasGap(SH2[free])} <->
{EGFR(Y992{p}[1]), RasGap(SH2[1])} @ [1,1]

rule EGFR_Grb2 {EGFR(Y1068{p}[free]), Grb2(SH2[free])} <->
{EGFR(Y1068{p}[1]), Grb2(SH2[1])} @ [1,1]

rule Grb2_SoS {Grb2(SH3[free]), SoS(a[free], SS{u}[free])} ->
{Grb2(SH3[1]), SoS(a[1], SS{u}[free])} @ [1]

rule Grb2_SoS_op {Grb2(SH3[1]), SoS(a[1])} ->
{Grb2(SH3[free]), SoS(a[free])} @ [1]

rule EGFR_Shc {EGFR(Y1148{p}[free]), Shc(PTB[free])} <->
{EGFR(Y1148{p}[1]), Shc(PTB[1])} @ [1,1]

rule Shc_Grb2 {Shc(Y318{p}[free]), Grb2(SH2[free])} <->
{Shc(Y318{p}[1]), Grb2(SH2[1])} @ [1,1]

rule Shc_at_318 {EGFR(CR[bound], Y1148{p}[1]), Shc(PTB[1],
Y318{u}[free])} -> {EGFR(CR[bound], Y1148{p}[1]), Shc(PTB[1],
Y318{p}[free])} @ [1]

rule Shc_at_318_op {Shc(Y318{p}[free])} -> {Shc(Y318{u}[free])} @ [1]

// Activating Ras
// #activate:
rule long_arm_SoS_Ras {EGFR(Y1148{p}[1]), Shc(PTB[1], Y318{p}[2]),
Grb2(SH2[2], SH3[3]), SoS(a[3], b[free]), Ras(S1S2{gdp}[free])} ->
{EGFR(Y1148{p}[1]), Shc(PTB[1], Y318{p}[2]), Grb2(SH2[2], SH3[3]),
SoS(a[3], b[4]), Ras(S1S2{gdp}[4])} @ [1]

rule short_arm_SoS_Ras {EGFR(Y1068{p}[1]), Grb2(SH2[1], SH3[2]),
SoS(a[2], b[free]), Ras(S1S2{gdp}[free])} -> {EGFR(Y1068{p}[1]),
Grb2(SH2[1], SH3[2]), SoS(a[2], b[3]), Ras(S1S2{gdp}[3])} @ [1]

rule Ras_GTP {SoS(b[1]), Ras(S1S2{gdp}[1])} ->
{SoS(b[1]), Ras(S1S2{gtp}[1])} @ [1]

rule SoS_Ras_op {SoS(b[1]), Ras(S1S2[1])} ->
{SoS(b[free]), Ras(S1S2[free])} @ [1]

// #deactivate:
rule direct_RasGap_Ras {EGFR(Y992{p}[1]), RasGap(SH2[1], s[free]),
Ras(S1S2{gtp}[free])} -> {EGFR(Y992{p}[1]), RasGap(SH2[1], s[2]),
Ras(S1S2{gtp}[2])} @ [1]

rule Ras_GDP {RasGap(s[1]), Ras(S1S2{gtp}[1])} ->
{RasGap(s[1]), Ras(S1S2{gdp}[1])} @ [1]

rule RasGAP_Ras_op {RasGap(s[1]), Ras(S1S2[1])} ->
{RasGap(s[free]), Ras(S1S2[free])} @ [1]

rule intrinsic_Ras_GDP {Ras(S1S2{gtp}[free])} ->
```

```
{Ras(S1S2{gdp}[free])} @ [1]

// Activating Raf
// #activation:
var p_Raf = {Ras(S1S2{gtp}[1]), Raf(x{u}[1])}
rule Ras_Raf {Ras(S1S2{gtp}[free]), Raf(x{u}[free])} -> p_Raf @ [1]
rule Raf p_Raf -> {Ras(S1S2{gtp}[1]), Raf(x{p}[1])} @ [1]
rule Ras_Raf_op {Ras(S1S2{gtp}[1]), Raf(x[1])} ->
{Ras(S1S2{gtp}[free]), Raf(x[free])} @ [1]

// #deactivation
var p_Raf2 = {PP2A1(s[1]), Raf(x{p}[1])}
rule PP2A1_Raf {PP2A1(s[free]), Raf(x{p}[free])} -> p_Raf2 @ [1]
rule Raf_op p_Raf2 -> {PP2A1(s[1]), Raf(x{u}[1])} @ [1]
rule PP2A1_Raf_op {PP2A1(s[1]), Raf(x[1])} ->
{PP2A1(s[free]), Raf(x[free])} @ [1]

// Activating MEK
// #activation:
var p_MEK = {Raf(x{p}[1]), MEK(S222{u}[1])}
rule Raf_MEK_at_222 {Raf(x{p}[free]), MEK(S222{u}[free])} ->
p_MEK @ [1]

rule MEK_at_222 p_MEK -> {Raf(x{p}[1]), MEK(S222{p}[1])} @ [1]
rule Raf_MEK_at_222_op {Raf(x{p}[1]), MEK(S222[1])} ->
{Raf(x{p}[free]), MEK(S222[free])} @ [1]

var p_MEK2 = {Raf(x{p}[1]), MEK(S218{u}[1])}
rule Raf_MEK_at_218 {Raf(x{p}[free]), MEK(S218{u}[free])} ->
p_MEK2 @ [1]

rule MEK_at_218 p_MEK2 -> {Raf(x{p}[1]), MEK(S218{p}[1])} @ [1]
rule Raf_MEK_at_218_op {Raf(x{p}[1]), MEK(S218[1])} ->
{Raf(x{p}[free]), MEK(S218[free])} @ [1]

// #deactivation:
var p_MEK3 = {PP2A2(s[1]), MEK(S222{p}[1])}
rule PP2A2_MEK_at_222 {PP2A2(s[free]), MEK(S222{p}[free])} ->
p_MEK3 @ [1]

rule MEK_at_222_op p_MEK3 -> {PP2A2(s[1]), MEK(S222{u}[1])} @ [1]
rule PP2A2_MEK_at_222_op {PP2A2(s[1]), MEK(S222[1])} ->
{PP2A2(s[free]), MEK(S222[free])} @ [1]

var p_MEK4 = {PP2A2(s[1]), MEK(S218{p}[1])}
rule PP2A2_MEK_at_218 {PP2A2(s[free]), MEK(S218{p}[free])} ->
p_MEK4 @ [1]

rule MEK_at_218_op p_MEK4 -> {PP2A2(s[1]), MEK(S218{u}[1])} @ [1]
rule PP2A2_MEK_at_218_op {PP2A2(s[1]), MEK(S218[1])} ->
{PP2A2(s[free]), MEK(S218[free])} @ [1]

// Activating ERK
// #activation:
var p_ERK = {MEK(s[1], S218{p}[free], S222{p}[free]), ERK(T185{u}[1])}
rule MEK_ERK_at_185 {MEK(s[free], S218{p}[free], S222{p}[free]),
ERK(T185{u}[free])} -> p_ERK @ [1]

rule ERK_at_185 p_ERK -> {MEK(s[1], S218{p}[free], S222{p}[free]),
ERK(T185{p}[1])} @ [1]

rule MEK_ERK_at_185_op {MEK(s[1]), ERK(T185[1])} -> {MEK(s[free]),
ERK(T185[free])} @ [1]

var p_ERK2 = {MEK(s[1], S218{p}[free], S222{p}[free]), ERK(Y187{u}[1])}
rule MEK_ERK_at_187 {MEK(s[free], S218{p}[free], S222{p}[free]),
ERK(Y187{u}[free])} -> p_ERK2 @ [1]

rule ERK_at_187 p_ERK2 -> {MEK(s[1], S218{p}[free], S222{p}[free]),
ERK(Y187{p}[1])} @ [1]

rule MEK_ERK_at_187_op {MEK(s[1]), ERK(Y187[1])} -> {MEK(s[free]),
ERK(Y187[free])} @ [1]

// #deactivation
var p_ERK3 = {MKP3(s[1]), ERK(T185{p}[1])}
rule MKP_ERK_at_185 {MKP3(s[free]), ERK(T185{p}[free])} -> p_ERK3 @ [1]
rule ERK_at_185_op p_ERK3 -> {MKP3(s[1]), ERK(T185{u}[1])} @ [1]
rule MKP_ERK_at_185_op {MKP3(s[1]), ERK(T185[1])} -> {MKP3(s[free]),
ERK(T185[free])} @ [1]

var p_ERK4 = {MKP3(s[1]), ERK(Y187{p}[1])}
rule MKP_ERK_at_187 {MKP3(s[free]), ERK(Y187{p}[free])} -> p_ERK4 @ [1]
rule ERK_at_187_op p_ERK4 -> {MKP3(s[1]), ERK(Y187{u}[1])} @ [1]
rule MKP_ERK_at_187_op {MKP3(s[1]), ERK(Y187[1])} -> {MKP3(s[free]),
ERK(Y187[free])} @ [1]

// Deactivating SoS
rule SoS_ERK {SoS(SS{u}[free]), ERK(s[free], T185{p}[free],
Y187{p}[free])} -> {SoS(SS{u}[1]), ERK(s[1], T185{p}[free],
Y187{p}[free])} @ [1]

rule SoS_ERK_op {SoS(SS[1]), ERK(s[1])} -> {SoS(SS[free]),
ERK(s[free])} @ [1]

// #feedback creation
rule SoS_at_SS {SoS(SS{u}[1]), ERK(s[1], T185{p}[free],
Y187{p}[free])} -> {SoS(SS{p}[1]), ERK(s[1], T185{p}[free],
Y187{p}[free])} @ [1]

// #feedback recovery
rule SoS_at_SS_op {SoS(SS{p}[free])} -> {SoS(SS{u}[free])} @ [1]

// Initialization
init i_EGF 10 {EGF(r{ext}[free])}
init i_EGFR 100 {EGFR(L{ext}[free], CR[free], Y992{u}[free],
 Y1068{u}[free], Y1148{u}[free])}

init i_Shc 100 {Shc(PTB[free], Y318{u}[free])}
init i_Grb_SoS 100 {Grb2(SH2[free], SH3[1]),
SoS(a[1], b[free], SS{u}[free])}

init i_RasGap 200 {RasGap(SH2[free], s[free])}
init i_Ras 100 {Ras(S1S2{gdp}[free])}
init i_Raf 100 {Raf(x{u}[free])}
init i_PP2A1 25 {PP2A1(s[free])}
init i_PP2A2 50 {PP2A2(s[free])}
init i_MEK 200 {MEK(s[free], S222{u}[free], S218{u}[free])}
init i_ERK 200 {ERK(s[free], T185{u}[free], Y187{u}[free])}
init i_MKP3 50 {MKP3(s[free])}

// Observables
obs ERK_pp {ERK(Y187{p}[?], T185{p}[?])}
obs MEK_pp {MEK(S222{p}[?], S218{p}[?])}
obs Raf_p {Raf(x{p}[?])}
obs Ras_gtp {Ras(S1S2{gtp}[?])}

// Terminates
terminate t_it time=300000
```