Multi-Level Modeling and Simulation of Cellular Systems - An Introduction to ML-Rules*

Tobias Helms, Tom Warnke, and Adelinde M. Uhrmacher

University of Rostock, Institute of Computer Science,
Albert-Einstein-Straße 22, 18059 Rostock, Germany
{tobias.helms,tom.warnke,adelinde.uhrmacher}@uni-rostock.de
http://mosi.informatik.uni-rostock.de

Abstract. ML-Rules is a rule-based language for multi-level modeling and simulation. ML-Rules supports dynamic nesting of entities and applying arbitrary functions on entity attributes and content, as well as for defining kinetics of reactions. This allows describing and simulating complex cellular dynamics operating at different organizational levels, e.g., to combine intra-, inter- and cellular dynamics, like the proliferation of cells, or to include compartmental dynamics like merging and splitting of mitochondria or endocytosis. The expressiveness of the language is bought with additional efforts in executing ML-Rules models. Therefore, various simulators have been developed from which the user and automatic procedures can select. The experiment specification language SESSL facilitates design, execution, and reuse of simulation experiments. The chapter illuminates the specific features of ML-Rules as a rule-based modeling language, the implications for an efficient execution, and shows ML-Rules at work.

Keywords: computational biology, rule-based modeling, multi-level modeling, cell biological systems, stochastic simulation, experiment specification

Preprint

1 Introduction

One of the first tasks in executing a simulation study is determining the content of the simulation model to be developed (or selected), which typically includes requirements, model input, output, assumptions, or simplifications [33]. Therefore, it is necessary to understand the real system that is the subject of the simulation study as well as the methodological implications of the scientific questions to be answered by the in-silico experiments. However, to select suitable methods for the in-silico experiments, knowledge about methods, their respective constraints and features is required. Within this chapter, we will present some specific features

* This manuscript, a preprint version, was prepared for an upcoming volume in Springer's *Methods in Molecular Biology* series, entitled "**Modeling Biomolecular Site Dynamics: Methods and Protocols**" (W.S. Hlavacek, Editor)

that the modeling language ML-Rules [26, 27, 36] adds to the family of rule-based languages in cell biology.

The motivation to develop ML-Rules has been to support multi-level modeling, i.e., describing a system at different organizational levels and relating the dynamics of the different levels. Considering multiple levels appears essential for a better understanding of biological systems. This refers to ecological systems, where "explanation of observed behaviour is not possible with reference solely to the spatial-temporal scale at which the observation was made" [37], as well as to cell-biological systems that "tend to regulate themselves by feedback effects, that is, by a process in which higher-level (systems) parameters influence lower-level components" [29]. Many cell biological models, for example, focus on intra-cellular dynamics. However, those influence and are influenced by dynamics at cell level, e.g., the proliferation and differentiation of stem cells, and cell-cell interaction.

For capturing the hierarchical organization and the causalities between different levels, i.e., from the lower to the upper (upward causation) and vice versa (downward causation) [3], ML-Rules supports a hierarchical dynamic nesting of model entities and upward and downward causation between different levels. Describing each level with its properties and dynamics explicitly and relating those different levels requires a high flexibility of the language: arbitrary attributes can be assigned to model entities, and arbitrary functions help to assess and accessing these attributes, constraining the kinetics, as well as relating dynamics at different levels. Thus, ML-Rules supports multi-level modeling by augmenting the well-established rule schemata with explicit dynamic hierarchical nesting of model entities, assigning attributes and content to model entities at each level. Rules are defined and applied on nested model entities as well as model entities that are nested within others.

Thus, already on first view certain differences to other rule-based modeling approaches exist, e.g., in comparison to BioNetGen [2] and Kappa [8] which focus on molecular binding of species to complexes. These differences, along with the features of ML-Rules, will be detailed in the following sections.

In section 2, the most important features of ML-Rules are illustrated with simple example models. The models are intentionally kept as simple as possible, focusing on the presented features. We included the models without any claim of them being biologically meaningful; for realistic applications of ML-Rules we refer to other sources, e.g., [15]. In Section 3, we discuss computational challenges caused by the expressive features of ML-Rules and clarify how these features influence the runtime performance of the simulator. Finally, in Section 4, we demonstrate how the simulation experiment specification language SESSL can be used to perform simulation experiments with ML-Rules models, e.g., parameter fitting by optimization.

The implementation of ML-Rules is open source and available in our source code repository at `https://git.informatik.uni-rostock.de/mosi/mlrules2`. Besides, a sandbox editor is available to create and execute models.

Preprint

```
// constants
k1: 1e-3; k2: 2; k3: 1; k4: 10; k5: 0.1;
n: 1000;

// species definitions
E(); // enzyme
S(); // substrat
ES(); // enzyme-substrat complex
P(); // product
EP(); // enzyme-product complex

// initial solution
>>INIT[n E + n S];

// rules
E:e + S:s -> ES @ k1 * #e * #s;
ES:es -> E + S @ k2 * #es;

ES:es -> EP @ k3 * #es;

EP:ep -> E + P @ k4 * #ep;
E:e + P:p -> EP @ k5 * #e * #p;
```

Fig. 1. A simple *enzyme-substrat-product* model written in ML-Rules.

Preprint

2 Modeling in ML-Rules

Starting with a simple example model to present the basic syntax of ML-Rules (see Section 2.1), this section step-by-step introduces the most important features of ML-Rules, i.e., attributed species and rule variables (see Section 2.2), dynamic compartments and multi-level rules (see Section 2.3), and functions on solutions (see Section 2.4). Simple models are given to illustrate the individual features. All listings show complete and valid ML-Rules models that can directly be used to execute simulations.

2.1 An Introductory Example

Figure 1 shows a complete ML-Rules implementation of a simple model representing an *enzyme-substrate-product* network. In this network, enzymes (`E`) and substrates (`S`) can form fragile *enzyme-substrate* complexes (`ES`). Such an *enzyme-substrate* complex can transform the substrate to a product (`P`) and form a fragile *enzyme-product* complex (`EP`). Enzymes and products can again form *enzyme-product* complexes.

The ML-Rules implementation of this model begins with definitions of constants (ll. 2-3). Constants can be used for example to calculate reaction rates or to set the initial amount of species. A name of a constant must always start with

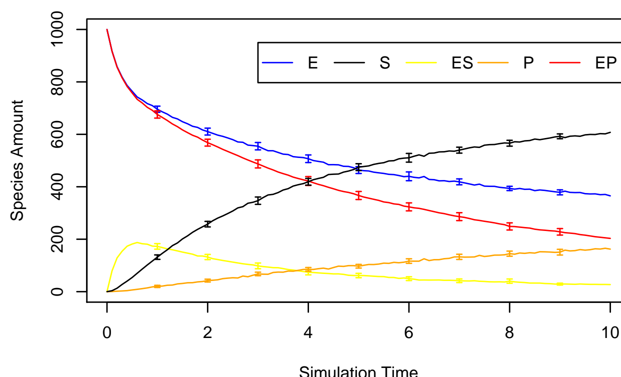

Fig. 2. Simulation results of the *enzyme-substrate-product* model with the stochastic simulator based on 20 replications.

Preprint

a lower letter. Next, species are defined (ll. 6-10). Species names must always start with a capital letter. Attributes can be defined within parentheses after the species name (see Section 2.2). Since this model does not use attributes, the parentheses are empty. In line 13, the initial entity multiset called *initial solution* (in the following, a multiset of entities is called *solution*) is defined, which contains `1000` entities of the species `E` and `1000` entities of the species `S` (`n = 1000`). The `+` operator is used to connect entities to a solution. Finally, the rules of the model are defined (ll. 16-22). Rules consist of three parts: reactants, products and a kinetic rate:

```
reactants -> products @ rate
```

For example, the first rule (l. 16) describes the process that an enzyme (`E`) and a substrate (`S`) form a complex (`ES`). Like in most rule-based languages, a rule is a pattern for a set of chemical reactions, i.e., based on the state of the system, a rule can result in various chemical reactions [2]. In this model, however, every rule exactly results in one reaction resulting in a reaction network with five reactions. All rules apply the law of mass action, i.e., the rate of a rule depends on a constant multiplied with the amounts of the reactants. The amount of a reactant can be accessed via reactant variables and the `#` operator. Reactant variables have to be defined after a reactant separated by a colon. For example, `e` and `s` are reactant variables of the first rule in line 16 and therefore `#e` and `#s` return the amount of the enzymes and substrates.

The standard simulator for executing ML-Rules models is based on the stochastic simulation algorithm (Direct Method) [14], where each reaction is calculated individually. A stochastic simulation is beneficial if stochastic effects influence the model behavior, e.g., in case species with small amounts are considered, which is easily the case if compartments and cells make up part of the species and dynamics to be considered. Figure 2 shows simulation results of the

model calculated with the stochastic simulator. Due to stochastic effects one simulation run does not suffice, but multiple replications are required to assess the possible behavior. More details about simulation algorithms and a discussion about possible improvements are given in Section 3.

2.2 Attributed Species and Reactant Variables

Attributed species are an essential concept of rule-based modeling languages [2, 8, 30, 21]. Attributes allow introducing variables in reactant patterns and therefore enable succinct model implementations. ML-Rules supports attributed species with attributes of the following types:

1. `bool`: `true` or `false`
2. `num`: real numbers
3. `string`: character sequences
4. `link`: `free` or unique binding values

Figure 3 shows a model of an abstract cell cycle using an attributed species. In line 5, one species `Cell` is defined with three attributes: a numerical attribute for the volume, a string attribute for the state of the cell (`G1`, `SG2`, `M`), and a boolean attribute representing the growth activity of the cell (enabled (`true`) or disabled (`false`)). In the definition of a species, the number of attributes of every entity of this species is fixed, i.e., every entity of the species `Cell` has always exactly these three attributes. Attributes do not have explicit names in ML-Rules, but are always determined by their position in the attribute tuple of a species. This design choice works well if species do not have too many attributes[1]. The initial solution (l. 8) consists of ten cells with the volume `1.0`, the state `'G1'`, and the growth activity enabled (`true`).

The first rule (ll. 11-12) describes the growth of a cell. The reactant pattern `Cell(vol,state,true)` uses two attribute variables `vol` and `state`. It matches all entities with an arbitrary volume and state, but the growth activity must be enabled. For example, given the initial solution, the simulator finds one match for the reactant pattern and creates one reaction with `vol = 1.0` and `state = 'G1'`. The attribute variables are reused in the product to describe a cell with the same state and an increased volume. Arithmetical expressions and functions can directly be used to calculate attribute values of products. Here, the function `unif(0,1)`, which returns a sample from the uniform distribution $U(0,1)$, is used to increase the current volume of the reactant cell. A comprehensive list of all auxiliary functions is given in the ML-Rules manual.

The second rule (ll. 13-14) describes the change of a cell from the state `'G1'` to `'SG2'`. Here, the rate of the rule is influenced by the volume of a matched `Cell`, it must be greater than `2.0`. Analog, the third rule (ll. 15-16) describes the change of a cell from `'SG2'` to `'M'`. A cell division is described in the fourth rule (ll. 17-18). The daughter cells start again in the state `'G1'` with the same

[1] Other rule-based languages, e.g., ML-Space [1], opted for accessing attributes by names.

Preprint

```
// constants
k1: 0.1; k2: 0.5; k3: 0.4; k4: 0.3; k5: 0.2;

// species definitions
Cell(num,string,bool);

// initial solution
>>INIT[10 Cell(1.0,'G1',true)];

// rules
Cell(vol,state,true):c -> Cell(vol+unif(0,1),state,true)
        @ k1 * #c;
Cell(vol,'G1',active):c -> Cell(vol,'SG2',active)
        @ if (vol > 2.0) then k2 * #c else 0;
Cell(vol,'SG2',active):c -> Cell(vol,'M',active)
        @ k3 * #c;
Cell(vol,'M',active):c -> 2 Cell(vol/2,'G1',active)
        @ k4 * #c;
Cell(vol,state,active):c -> Cell(vol,state,!active)
        @ k5 * #c;
```

Fig. 3. An excerpt of the a simple *cell cycle* model written in ML-Rules.

Preprint

growth activity as the parent cell and with its halved volume. Finally, the last rule (ll. 19-20) describes a change of the growth activity. For boolean variables, analogous to many programming languages, "`!`" denotes the negation.

Altogether, this model describes an infinite reaction network, because the set of possible assignments for the volume attribute is infinite. Therefore, for this model the complete reaction network cannot be calculated once at the beginning of a simulation run. Only a part of the network based on the current state can be computed, which must be updated frequently resulting in computational overhead, see Section 3.

The attribute type `link` can be used to model species complexes without enumerating all possible complexes explicitly. The simple model described in Section 2.1 uses explicit species to represent *enzyme-substrate* complexes (`ES`) and *enzyme-product* complexes (`EP`). However, listing all possible combinations of complexes becomes impracticable if the number of possible complexes increases, e.g., in case a model contains species with various binding sites. Coping effectively with these multi-state biomolecules has been one of the driving forces behind rule-based modeling approaches [11]. Accordingly, explicit species definitions for such complexes can be avoided in ML-Rules by using attributes of species and the attribute type `link`. Figure 4 shows the adaptation of the *enzyme-substrate-product* model using link attributes. All species have one attribute of type `link` representing one binding site (ll. 6-8). Each attribute of the type `link` has either the value `free` to represent the unbound state or a key representing the binding

```
1  // constants
2  k1: 1e-3; k2: 2; k3: 1; k4: 10; k5: 0.1;
3  n: 1000;
4
5  // species definitions
6  E(link); // enzyme with one binding site
7  S(link); // substrat with one binding site
8  P(link); // product with one binding site
9
10 // initial solution
11 >>INIT[n E(free) + n S(free)];
12
13 // rules
14 E(free):e + S(free):s -> E(x) + S(x)
15         @ k1 * #e * #s where x = nu();
16 E(x) + S(x) -> E(free) + S(free)
17         @ if (x != free) then k2 else 0;
18
19 E(x) + S(x) -> E(x) + P(x)
20         @ if (x != free) then k3 else 0;
21
22 E(x) + P(x) -> E(free) + P(free)
23         @ if (x != free) then k4 else 0;
24 E(free):e + P(free):p -> E(x) + P(x)
25         @ k5 * #e * #p where x = nu();
```

Fig. 4. The simple *enzyme-substrat-product* model (see Section 2.1) using links to describe the complexes.

between entities containing the same key. For example, the first rule (ll. 14-15) takes one unbound enzyme (`E(free)`) and one unbound substrate (`S(free)`) and connects them by assigning the same link value to their first attribute. A new unique link value is generated by the method `nu()`. Since the same link value shall be used twice to assign it to both product entities, the variable `x` is defined at the end of the rule in a `where` clause. The `where` clause can be used to create variables for rules and is inspired by the `where` clause of the Haskell programming language [22]. The rules dealing with bound entities (ll. 16–23) use the same reactant variable `x` for two reactants, which means that the variable must have the same value for both reactants. Consequently, exactly one pair of one enzyme and one substrate can match both reactants for example of the rule unbinding them (ll. 16-17). Since each link value is unique, all bound enzymes, substrates and products are treated as individuals. Thus, their amounts do not have to be considered in the rate equations. Realizing binding by new values is adapted from process algebras, where new names are generated to allow a private communication between individual processes over a private channel [32]. Whereas the forming of complexes by entities that share “private” attribute values

Preprint

```
// constants
k1: 0.02;
k2: 0.15;
nBCatCell: 10000;
nBCatNuc: 4000;

// species definitions
Cell()[]; // compartment
Nucleus()[]; // compartment
BCat();

// initial solution
>>INIT[3 Cell[1 Nucleus[nBCatNuc BCat] + nBCatCell BCat]];

// rules
Nucleus[s?] + BCat:b -> Nucleus[BCat + s?] @ k1 * #b;
Nucleus[BCat:b + s?] -> Nucleus[s?] + BCat @ k2 * #b;
```

Fig. 5. A model of β-catenin proteins (BCat) shuttling into and out of the nucleus of a cell.

avoids introducing a further construct to the modeling language, bound entities in ML-Rules constitute individuals due to the unique values of the attributes. This implies that for each bound species pair, the simulator creates one reaction for the reaction network resulting in a much larger network compared to the reaction network of the implementation in Section 2.1 and moreover, this network again has to be frequently updated during a simulation. This is the reason why other languages, e.g., Kappa, Bionetgen or ML-Space, introduced specific operators for binding.

2.3 Compartments and Dynamic Nestings

ML-Rules has been explicitly developed to support dynamically nested entities and multi-level rules. Figure 5 shows an ML-Rules implementation of a hierarchical model, in which β-catenin proteins shuttle into and out of the nucleus. Such processes are crucial for various pathways, e.g., the Wnt/β-catenin signaling pathway [28]. Although rather abstract, the model exploits several multi-level features of ML-Rules. First, compartments are defined marked with brackets behind the parentheses of a species definition (ll. 8-9). Only compartments are allowed to contain entities. Brackets are always used to describe nestings. Second, the description of the initial solution (l. 13) defines a nested solution: three cells are defined, each containing 10000 β-catenin proteins in its cytosol and containing 4000 β-catenin proteins in its nucleus. The first rule (l. 16) describes the shuttling of a β-catenin protein into a nucleus and the second rule (l. 17) out of a nucleus. The rules employ so-called *rest solution* variables (`<name>?`). These

Preprint

variables bind to the whole solution of a compartment except entities bound to other reactants. For example, the rest solution `s?` of the reactant

```
Nucleus[BCat:b + s?]
```

represents the whole content of a nucleus except one β-catenin entity. No reactant variables are used for the compartments, because they are currently treated individually in ML-Rules, i.e., the amount of a compartment is 1, as the contents of compartments typically vary. Since three cells are defined in the initial solution, both rules map to three reactions resulting in a reaction network with six reactions.

The described model presents a system with a static nesting structure. However, ML-Rules also allows creating, changing, and destroying compartments by rules. Figure 6 shows an ML-Rules implementation of an abstract endocytosis model exploiting dynamic compartments with the following behavior. A particle (`Particle`) can enter a cell (`Cell`) by forming an endosome compartment (`Endo`) in the cell containing the particle. Two endosomes can fuse, i.e., a new endosome is created that contains the content of both fused endosomes. Finally, endosomes can fuse with existing lysosomes (`Lyso`) of the cell — the endosome is destroyed and all content of the endosome is transferred to the lysosome.

The first rule (l. 15) describes the creation of an endosome containing the particle. The second rule (l. 16) describes the fusion of two endosomes by creating a new endosome containing the rest solutions of both reactant endosomes. The third rule (l. 17) follows the same idea as the second rule, but instead of two endosomes, one endosome and one lysosome merge to one lysosome. The concrete reaction network described by the three rules depends on the current state of the system, i.e., the reaction network is not fixed, but changes during a simulation run. Therefore, analogous to infinite reaction networks caused by continuous attributes of species, the simulator only computes a part of the reaction network based on the current state and updates it regularly as needed.

The combination of compartments and attributed species is illustrated in Figure 7, which shows a model using a numerical attribute to describe the volume of a compartmental cell `Cell`. For this, the species definition of the cell assigns a numerical attribute `num` to it (l. 9). Further, three species are defined that represent proteins important for the cell cycle [35]: `Cyclin` representing the cyclin protein, `CDK` representing a cyclin-dependent kinase, and `MPF` representing the maturation-promoting factor composed of one cyclin and one cyclin-dependent kinase.

Initially, the volume of all cells is 1.0 (l. 15). The first rule (ll. 18-19) describes the growth process of a cell. It uses an attribute variable: `vol`. During the simulation, the simulator matches the reactant pattern `Cell(vol)[s?]` to concrete `Cell` entities and uses their attribute values for this variable. Therefore, one potential reaction is created for each `Cell` entity. The attribute variable `vol` is also used to restrict the rule, i.e., it can only be applied to cells whose volume is smaller than `max`. The second rule uses the attribute variable `vol` to adapt the kinetic rate of this reaction, i.e., the larger the cell, the less likely the reaction. Finally, the third rule simply describes the degradation of `MPF` to a `Cyclin` and a `CDK`.

Preprint

```
// constants
k1: 0.001;
k2: 0.002;

// species definitions
Cell()[]; // compartment
Endo()[]; // compartment
Lyso()[]; // compartment
Particle();

// initial solution
>>INIT[100 Particle + 3 Cell[5 Lyso]];

// rules
Cell[s?] + Particle:p -> Cell[Endo[Particle] + s?] @ k1 * #p;
Endo[s1?] + Endo[s2?] -> Endo[s1? + s2?] @ k2;
Endo[s1?] + Lyso[s2?] -> Lyso[s1? + s2?] @ k2;
```

Fig. 6. An abstract endocytosis model illustrating the creation and fusion of compartments.

Preprint

2.4 Developing Complex Rules with Functions on Solutions

In some cases, modelers want to describe complex behavior that cannot be easily realized with the presented features of ML-Rules like attributed species, dynamic nestings, or multi-level rules. For example, although compartments can easily be merged, how to split one compartment into two compartments? Further, how to describe a significant change of the whole system with one rule, e.g., instantaneous kill 10% of all cells due to the execution of a treatment?

Functions on solutions allow modelers to describe such complex phenomena. They deal with solutions as input parameters. These input solutions can be analyzed to compute some statistics required, e.g., calculate the average volume of all existing cells. In addition, functions on solutions can also calculate an output solution, e.g., how the content of a cell shall be split between its daughter cells. ML-Rules provides a library of basic functions on solutions to realize common tasks, e.g., to count a species in a solution or to remove one species from a solution. A detailed list of library functions, which is regularly extended, is available in the ML-Rules manual. However, we also added means — inspired by functional programming — to ML-Rules which allow users implementing their own functions on solutions within the model file.

Figure 8 shows a simple model using functions on solutions to realize the partitioning of species during a cell division. The rule (ll. 17-19) takes a cell and creates two new cells, whereby the content of the original cell represented by the rest solution `sol?` is split by calling the function `split` in the `where` part of the rule. A cell can only split if the number of proteins within this cell is greater than

```
1  // constants
2  k1: 0.001;
3  k2: 0.002;
4  k3: 0.004;
5  max: 3;
6  gf: 0.01;
7
8  // species definitions
9  Cell(num)[]; // cell with numerical volume
10 Cyclin();
11 CDC2();
12 MPF();
13
14 // initial solution
15 >>INIT[100 Cell(1.0)[100 CDC2 + 100 Cyclin]];
16
17 // rules
18 Cell(vol)[s?] -> Cell(vol+gf)[s?]
19         @ if (vol < max) then k1 else 0;
20 Cell(vol)[CDC2:cdc + Cyclin:cyc + s?] -> Cell(vol)[MPF + s?]
21         @ k2 * #cdc * #cyc / vol;
22 Cell(vol)[MPF:m + s?] -> Cell(vol)[CDC2 + Cyclin + s?]
23         @ k3 * #m;
```

Fig. 7. A simple model illustrating changes in attributes and content.

Preprint

5 (l. 18). A tuple `<left,right>` is defined (l. 19), so that the variables `left` and `right` represent both partitions of the original content assigned to `sol?`.

The function `split` is defined in the beginning of the model (ll. 1-7). Initially, the type declaration of the function is given (l. 1): it has a solution as a parameter and returns a tuple. The type declaration in ML-Rules reads like:

```
function name :: parameter type -> ... -> result type
```

Next, pattern matching of parameters is used to define different cases of a function — analogous to pattern matching used in Haskell. The first case of `split` (l. 2) is used when the function is called with an empty solution (denoted by empty brackets). In this case, the function simply returns a tuple containing two empty solutions. The second case (l. 4-7) describes the function when called with a solution at least containing one entity. The parameter solution is written as `x + xs`, whereby `x` is an arbitrary entity of the solution and `xs` is the rest of the parameter solution. Three variables are calculated (ll. 5-7) to determine the result tuple of this case. The variable `numl` represents the halved amount of current entity `x` rounded down. The variable `numr` represents the other half of the amount of `x`. For example, if `x = 81 Particle`, then `numl = 40` and `numr = 41`. The tuple `<restl,restr>` represents the partitioning of the solution `xs`,

```
split :: sol -> tuple;
split []     = <[],[]>;
split x + xs = <new(numl,name(x),att(x)) + restl,
                new(numr,name(x),att(x)) + restr>
    where numl          = round(amount(x)*0.5),
          numr          = amount(x) - numl,
          <restl,restr> = split(xs);

//species
Cell()[];
Protein();

//initial solution
>>INIT[10 Cell[100 Protein]];

//reaction rules
Cell[sol?] -> Cell[left] + Cell[right]
    @ if (count(sol?,'Protein') > 5) then 1 else 0
    where <left,right> = split(sol?);
```

Fig. 8. Example of functions on solutions.

i.e., here the recursive step is executed. By using these three variables, the result of the function is a tuple, whereby the left (right) solution of the tuple contains the recursively calculated left (right) partitioning of `xs` and the species `x` with the amount `numl` (`numr`). The function `new` creates an entity with the given parameter: the first parameter represents the amount of the calculated entity, the second parameter represents the name of the calculated entity, and the third parameter represents the attributes of the calculated entity.

We are aware that functions on solutions are a complex concept and modelers who are not familiar with functional programming might be discouraged to implement their own functions. Therefore, our aim is to extend the functions library regularly to provide as many useful functions as possible.

3 The challenge of efficient simulations with ML-Rules

The standard simulator that ML-Rules is shipped with (and most of its variants) base on the *stochastic simulation algorithm* (SSA) [14]. For a vector of chemical species amounts

$$\mathbf{X} = (x_1, x_2, \ldots, x_n) \in \mathbb{N}^n$$

and a set of chemical reactions

$$\mathbf{R} = \{R_1, R_2, \ldots, R_m\},$$

where a chemical reaction R_i is characterized by a change vector

$$v_i = (v_{i1}, v_{i2}, \ldots, v_{in}) \in \mathbb{Z}^n$$

Preprint

and a kinetic rate function $a_i : \mathbb{N}^n \rightarrow \mathbb{R}^+$, the SSA computes trajectories by iteratively calculating the following steps:

1. Compute the sum of all kinetic rates: $a_{sum}(\mathbf{X}) = \sum_{i=1}^{m} a_i(\mathbf{X})$.
2. Select one reaction to be fired. The probability $P(R_i)$ to select a reaction R_i is its relation of the kinetic rate $a_i(\mathbf{X})$ to $a_{sum}(\mathbf{X})$:
$$P(R_i) = \frac{a_i(\mathbf{X})}{a_{sum}(\mathbf{X})}.$$
Thus, the selection method finds the smallest index i so that
$$\sum_{i=1}^{m} a_i(\mathbf{X}) > x,$$
where x is sampled from the uniform distribution $U(0, a_{sum}(\mathbf{X}))$.
3. The selected reaction R_i is executed, i.e., $\mathbf{X} := \mathbf{X} + v_i$.
4. Finally, the simulation time is advanced by sampling a number from an exponential distribution with rate $\lambda = a_{sum}(\mathbf{X})$:
$$t := t + Exp(a_{sum}(\mathbf{X})).$$

Applied to ML-Rules, the SSA has to be extended by an additional initial step: The reaction set $\mathbf{R}$ has to be calculated. This has to be done since the reaction set $\mathbf{R}$ is dynamic in ML-Rules for example due to dynamic structures and therefore it is in general not sufficient to compute it once at the beginning of the simulation, e.g., as it can be done in BioNetGen. Calculating the reaction set usually causes most of the computational costs to simulate an ML-Rules model. Even if $\mathbf{R}$ is only updated where necessary, i.e., only invalid reactions are removed from $\mathbf{R}$ and new reactions are added to $\mathbf{R}$, e.g., done efficiently by using a dependency graph [13], the computational costs of the reaction set update still dominate the load of an ML-Rules simulation. Nevertheless, for some ML-Rules models, the reaction set might be fixed, so that the update is not necessary and it can be avoided. For this case, we developed a tailored ML-Rules simulator which performs significantly better for such models.

To calculate all reactions for a given rule, the simulator has to find all entities for each reactant pattern that match this pattern. In ML-Rules, this pattern matching can be a complex task in case functions on solutions are used, e.g., if they iterate over solutions to calculate results. Particularly functions implemented in the model file can decrease the runtime performance, because these functions are so far not translated to Java code and compiled by the Java compiler for an efficient execution, but they are kept in a symbolic representation that has to be interpreted each time such a function is called. Transforming this symbolic representation to native Java code that can be used more efficiently by the ML-Rules simulator is part of future work. Besides, the so-called rigidity property (Lemma 3 in [6]) does not hold generally in ML-Rules, see also the discussion about the rigidity property referring to React(C) in [21]. This property

Preprint

implies that after matching one reactant pattern of a rule containing connected reactants, the remaining matching process becomes clearly determined, i.e., for each remaining reactant pattern, at most one concrete entity matches. Exploiting this property can simplify the matching process [7]. In ML-Rules, all entities enclosed in the same compartment are connected via this compartment and all reactants are connected since they assume matched entities to be enclosed by the same compartment. Nevertheless, for such connected reactants, the rigidity property does not hold in general.

Besides the calculation of the reaction set $\mathbf{R}$ itself, the size of $\mathbf{R}$ is an important factor influencing the performance of reaction network simulations. For example, in case of attributed species, huge reaction networks can easily be required exponentially growing with the number of attributes per species. To deal with this problem, network-free approaches have been developed that avoid an explicit calculation of $\mathbf{R}$, e.g., NFsim for BioNetGen [34]. NFSim treats every entity of every species individually, i.e., the state of the system is a set of individuals. Similarly, analog to Rete algorithms [12], the algorithm links every individual to every reactant that it matches to. Next, the number of links to each reactant is used to calculate the number of reactions that would be created if the reaction set $\mathbf{R}$ would be calculated. For example, if 10 individuals match to the first reactant of a rule and 20 individuals match to the second reactant of the rule, $10 \cdot 20 = 200$ reactions are possible between these individuals. This number is used to calculate the kinetic rate of a rule by simply multiplying it with the rate constant of the rule. Therefore, it is assumed that each combination of matched individuals result in the same kinetic rate, i.e., in particular this implies that rates are not allowed to depend on reactant variables. All kinetic rates of the rules are then used like in the SSA, i.e., the sum of all rates is computed, one rule is selected to be fired, and the simulation time is advanced based on the sum of rates. The only difference is that the selected rule cannot be executed directly, but concrete individuals for the reactants have to be selected first. After executing a rule with selected individuals, the links of individuals to reactants have to be updated properly. Altogether, this approach avoids the creation of the reaction set $\mathbf{R}$. A similar approach is used by the simulator of Bigraphs, which also avoids the calculation of $\mathbf{R}$ [23].

Network-free approaches perform particularly well if reactants include many links or attributes, i.e., the size of the rule set is much smaller than the size of the reaction set. Although the network-free approach can be beneficial also for ML-Rules models in principle, since kinetic rates often depend on reactant variables, all models we developed so far for ML-Rules seem not suitable for network-free simulation. There would simply be not enough links for the reactants for the approach to become beneficial. However, this might be different for models focusing on molecular binding of species to complexes. We have not implemented a network-free simulator for ML-Rules yet and further research is needed to evaluate the performance behavior of such a simulator for ML-Rules.

Instead of calculating every individual reaction separately, τ-leaping algorithms leap ahead calculating reactions that would have occured during the leap

Preprint

simultaneously. These approximate stochastic simulators trade accuracy for runtime efficiency. We developed a τ-leaping simulator for ML-Rules, which results in significant speed-ups for some models compared to the standard ML-Rules simulator [17].

In case of large species amounts when stochastic effects become negligible, chemical reaction networks can be simulated deterministically. For this deterministic approximation, the rules are transformed to differential equations and numerical integration methods are applied for the simulation. Therefore, the reaction network described by the rules of the model must be finite and fixed to calculate all differential equations. Models with continuous attributes or dynamic nestings cannot be computed purely deterministically — hybrid variants that treat some rules deterministically and some rules stochastically are needed, e.g., see [4, 9]. A hybrid simulator for ML-Rules models has also been developed for ML-Rules [19].

Altogether, the simulation of ML-Rules models is a complex task — often resulting in comparably slow simulation runs. To address this issue, several simulators have been and are still being developed for ML-Rules [18], well aware of the fact that the performance of simulators crucially depend on the model – an observation which has also motivated the construction of different simulators for other rule-based languages, e.g., BioNetGen [20]. Significant speedup can be achieved for subsets of models that do not use the complete set of features offered by ML-Rules, e.g., calculating kinetic rates based on current attribute values and content, as in those situations short-cuts taken by other rule-based languages are applicable. Approximate schemes are another possibility but the achieved speed-up varies with the model. Generally, it is rather difficult to predict the performance of simulators beforehand, here automatic selection procedures are needed [16].

4 Experiments with ML-Rules and SESSL

ML-Rules models do not contain information about their use in simulation experiments. It was a conscious decision to separate the model, i.e., the representation of the system at hand, from the simulation experiment, i.e., the methods that are involved in generating data from the model. Thereby, a central method is the method used for executing the model (Section 3), but for experimenting with a model more is needed: methods to initialize a model, to search the parameter space of the model, or to determine which outputs to observe, to name only a few [24]. By a clear separation of concern, different experimentation tools can be used to conduct ML-Rules simulation studies, and conversely these experimentation tools can employ different modeling languages. One tool that embodies this philosophy of decoupling software packages is the Simulation Experiment Specification via a Scala Layer (SESSL) [10].

SESSL is a domain-specific language that enables users to describe complex simulation experiments in a way that is declarative and directly executable by a computer. As an embedded language based on Scala, all parts of the experiment

Preprint

Line	Annotation
1	*Import SESSL core*
2	*Import ML-Rules binding*
3	
4	*Execute the following experiment*
5	*Include observation and parallelization*
6	
7	*Location and name of model file*
8	*Choose a simulation algorithm*
9	*Set parameter k1 to 0.005*
10	*Scan parameter k2 from 0.001 to 0.010*
11	*Use 4 threads*
12	*50 replications of each configuration*
13	*Simulation stops at time 1000*
14	*Observe the number of particles in endosomes*
14	*Observe at 0, 100, 200, . . . , 1000*
15	
16	*After each run*
17	*Output observed trajectory*
18	
19	
20	

```
import sessl._
import sessl.mlrules._

execute {
  new Experiment with Observation with
      ParallelExecution {
    model = "/data/models/endocytosis.mlrj"
    simulator = StandardSimulator()
    set("k1" <~ 0.005)
    scan("k2" <~ range(0.001, 0.001, 0.010))
    parallelThreads = 4
    replications = 50
    stopTime = 1000
    observe("endo-part" ~"Cell/Endo/Particle")
    observeAt(range(0, 100, 1000))

    withRunResult(results => {
      println(results~"endo-part")
    })
  }
}
```

Fig. 9. A SESSL experiment using ML-Rules (Scala keywords are shown in blue).

specification are valid program code. When executed, SESSL translates the described experiment to calls to a concrete simulation system, e.g., the ML-Rules simulator, and translates back the simulation output. Thus, experiment specifications can be reused with several simulation systems. The only change required is the binding, which manages the communication between SESSL and the simulation system. SESSL supports several configuration options of the simulation model and algorithm, e.g., model parametrization and observation, parallelization, and complex replication and stop conditions. Differences in the feature sets of different simulation systems are handled by the binding architecture. Besides bindings to simulation systems, SESSL experiments can also employ bindings for other tasks, such as output analysis, report generation, and simulation-based optimization. In any case, SESSL delegates most of the actual work to external software, thus forming a layer between users and heterogeneous simulation software packages.

Figure 9 shows an example SESSL specification for an experiment using ML-Rules. The simulation system to use is chosen by importing a binding (l.2). Line 5 demonstrates a further feature of SESSL: Experiments are configured by "mixing in" traits. This way, the feature set exploited by a concrete experiment is denoted and the Scala compiler is able to verify that the imported binding supports all feature traits that are mixed in. All bindings support the parametrization of the model and simulation configuration such as simulation stop time and replicating simulation runs. The binding for the ML-Rules simulation package includes additional feature traits to support parallel execution of simulation runs, complex replication conditions using confidence intervals, and also selective observation of model variables. When executed, a SESSL experiment using the ML-Rules binding initiates simulation runs with the specified configuration. For

Preprint

```
1  import sessl._
2  import sessl.optimization._
3  import sessl.mlrules._
4  import sessl.opt4j._
5
6  val ref = Seq[Double](18000, 15000, 11500) // reference values
7
8  minimize { (params, objective) =>
9    execute {
10     new Experiment with ParallelExecution with Observation {
11       model = "/data/models/bcat.mlrj"
12       simulator = SimpleSimulator()
13       parallelThreads = -1
14       replications = 5
15       stopTime = 25
16       set("k1" <~ params("k1")) // set model parameters as defined by optimizer
17       set("k2" <~ params("k2"))
18       observe("nucBCat" ~"Cell/Nucleus/BCat")
19       observeAt(0.0, 5.0, 20.0) // observation times of reference values
20       var runResults = 0.0
21       withRunResult(results => {
22         val numbers = results.values("nucBCat").asInstanceOf[Iterable[Double]]
23         runResults += Math.sqrt(mse(numbers, ref)) // calculate root mean square error
24       })
25       withReplicationsResult(results => {
26         objective <~ runResults / replications // store value of objective function
27         runResults = 0.0
28       })
29     }
30   }
31 } using new Opt4JSetup {
32   param("k1", 0.01, 0.01, 0.2) // optimization parameter bounds
33   param("k2", 0.01, 0.01, 0.2)
34   optimizer = ParticleSwarmOptimization(iterations = 40, particles = 30)
35   withOptimizationResults { results =>
36     println("Overall results: " + results.head)
37   }
38 }
```

Fig. 10. A SESSL experiment using ML-Rules and Opt4J (Scala keywords are shown in blue).

Preprint

example, if the SESSL specification sets a model parameter, the value of the homonymous constant in the ML-Rules file is overwritten.

The full power of SESSL becomes evident when an experiment combines the functionality of several bindings. Figure 10 shows an experiment employing simulation-based optimization for parameter fitting. A particle swarm optimization algorithm provided by Opt4J [25] is used to search for the parametrization of the model whose output most closely resembles a given sequence of reference values, for example obtained in wet-lab experiments. The functionality of SESSL can be augmented with custom user code, for example by invoking a function to calculate the mean square error (l. 23). Besides parameter fitting, SESSL experiments frequently employ statistical model checking [5]. Here SESSL's declarative nature comes in handy, e.g., by automatically generating experiments to support the process of developing models [31].

Using SESSL for specifying and executing ML-Rules experiments, instead of providing a user-interface to select experimentation methods, adds to the flexibility of executing experiments and tools to be used. SESSL provides the means to specify and reuse experiments similarly as models are specified and reused. The benefits of the approach are particularly evident for power-users as it facilitates the realization and documentation of problem tailored experiments. To ease also the regular modeler into using SESSL, we are currently developing a SESSL editor, and are steadily increasing the number of SESSL experiments to serve as a repository for future reuse.

5 Summary

This chapter gives an introduction to the rule-based modeling language ML-Rules. Important features of the language have been presented and illustrated based on simple model examples. The features that distinguish ML-Rules from other approaches is its ability to handle dynamic nesting, e.g., endocytosis or cell proliferation, and its support of arbitrary attributes and arbitrary function that can be used to access and change attributes and content of species, and to constrain the reactions. Thus, ML-Rules is a highly expressive language. This expressivity unfortunately does not come for free. The simulator of ML-Rules has to deal with infinite and dynamic reaction networks, resulting in regular, incremental updates of the used reaction network and thus computational overhead. Besides, we also showed how to execute complex simulation experiments with ML-Rules and SESSL.

Preprint

References

1. A. Bittig and A. M. Uhrmacher. ML-Space: Hybrid Spatial Gillespie and Particle Simulation of Multi-level Rule-based Models in Cell Biology. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2016. to appear.
2. M. L. Blinov, J. R. Faeder, B. Goldstein, and W. S. Hlavacek. BioNetGen: Software for Rule-based Modeling of Signal Transduction Based on the Interactions of Molecular Domains. *Bioinformatics*, 20(17):3289–3291, 2004.
3. D. T. Campbell. *'Downward Causation' in Hierarchically Organised Biological Systems*, pages 179–186. Macmillan Education UK, 1974.
4. Y. Cao, D. T. Gillespie, and L. R. Petzold. The slow-scale stochastic simulation algorithm. *The Journal of Chemical Physics*, 122(1), 2005.
5. E. M. Clarke, J. R. Faeder, C. J. Langmead, L. A. Harris, S. K. Jha, and A. Legay. Statistical Model Checking in BioLab: Applications to the Automated Analysis of T-Cell Receptor Signaling Pathway. In M. Heiner and A. M. Uhrmacher, editors, *Computational Methods in Systems Biology*, number 5307 in Lecture Notes in Computer Science, pages 231–250. Springer Berlin Heidelberg, 2008.
6. V. Danos, J. Feret, W. Fontana, and J. Krivine. *Scalable Simulation of Cellular Signaling Networks*, pages 139–157. Springer Berlin Heidelberg, 2007.
7. V. Danos, J. Feret, W. Fontana, and J. Krivine. Scalable Simulation of Cellular Signaling Networks. In *Proceedings of the 5th Asian Symposium on Programming Languages and Systems, APLAS*, pages 139–157, 2007.

8. V. Danos and C. Laneve. Formal molecular biology. *Theoretical Computer Science*, 325(1):69–110, 2004.
9. W. E, D. Liu, and E. Vanden-Eijnden. Nested stochastic simulation algorithm for chemical kinetic systems with disparate rates. *The Journal of Chemical Physics*, 123(19), 2005.
10. R. Ewald and A. M. Uhrmacher. SESSL: A Domain-specific Language for Simulation Experiments. *ACM Transactions on Modeling and Computer Simulation*, 24(2):11:1–11:25, Feb. 2014.
11. J. R. Faeder, M. L. Blinov, B. Goldstein, and W. S. Hlavacek. Rule-based modeling of biochemical networks. *Complexity*, 10(4):22–41, 2005.
12. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.
13. M. A. Gibson and J. Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *The Journal of Chemical Physics*, 104(9):1876–1889, 2000.
14. D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.
15. F. Haack, H. Lemcke, R. Ewald, T. Rharass, and A. M. Uhrmacher. Spatio-temporal Model of Endogenous ROS and Raft-Dependent WNT/Beta-Catenin Signaling Driving Cell Fate Commitment in Human Neural Progenitor Cells. *PLoS Computational Biology*, 11(3):e1004106, 2015.
16. T. Helms, R. Ewald, S. Rybacki, and A. M. Uhrmacher. Automatic Runtime Adaptation for Component-Based Simulation Algorithms. *ACM Transactions on Modeling and Computer Simulation*, 26(1):7:1–7:24, 2015.
17. T. Helms, M. Luboschik, H. Schumann, and A. M. Uhrmacher. An Approximate Execution of Rule-Based Multi-level Models. In *Proceedings of the 11th International Conference on Computational Methods in Systems Biology, CMSB*, pages 19–32, 2013.
18. T. Helms, T. Warnke, C. Maus, and A. M. Uhrmacher. Semantics and efficient simulation algorithms of an expressive multilevel modeling language. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 27(2):8:1–8:25, 2017.
19. T. Helms, P. Wilsdorf, and A. M. Uhrmacher. Hybrid Simulation of Dynamic Reaction Networks in Multi-Level Models. In *Proceedings of the 32nd Workshop on Principles of Advanced and Distributed Simulation (PADS'18)*, 2018.
20. J. S. Hogg, L. A. Harris, L. J. Stover, N. S. Nair, and J. R. Faeder. Exact Hybrid Particle/Population Simulation of Rule-Based Models of Biochemical Systems. *PLoS Computational Biology*, 10(4):1–16, 04 2014.
21. M. John, C. Lhoussaine, J. Niehren, and C. Versari. Biochemical Reaction Rules with Constraints. In *Proceedings of the 20th European Symposium on Programming, ESOP*, pages 338–357, 2011.
22. S. L. P. Jones. *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.
23. J. Krivine, R. Milner, and A. Troina. Stochastic Bigraphs. *Electronic Notes in Theoretical Computer Science*, 218:73–96, 2008.
24. S. Leye, J. Himmelspach, and A. M. Uhrmacher. A Discussion on Experimental Model Validation. pages 161–167. IEEE, 2009.
25. M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich. Opt4J: a modular framework for meta-heuristic optimization. page 1723. ACM Press, 2011.
26. C. Maus. *Toward Accessible Multilevel Modeling in Systems Biology: A Rule-based Language Concept.* PhD thesis, University of Rostock, 2013.

Preprint

27. C. Maus, S. Rybacki, and A. M. Uhrmacher. Rule-based multi-level modeling of cell biological systems. *BMC Systems Biology*, 5(166), 2011.
28. O. Mazemondet, M. John, S. Leye, A. Rolfs, and A. M. Uhrmacher. Elucidating the Sources of β-Catenin Dynamics in Human Neural Progenitor Cells. *PLoS ONE*, 7(8), 2012.
29. D. Noble. *The MUSIC of LIFE: Biology Beyond Genes*. Oxford University Press, 2008.
30. N. Oury and G. D. Plotkin. Multi-level modelling via stochastic multi-level multiset rewriting. *Mathematical Structures in Computer Science*, 23(2):471–503, 2013.
31. D. Peng, T. Warnke, F. Haack, and A. M. Uhrmacher. Reusing simulation experiment specifications to support developing models by successive extension. *Simulation Modelling Practice and Theory*, 2016 (to appear).
32. C. Priami. Stochastic π-Calculus. *The Computer Journal*, 38(7):578–589, 1995.
33. R. G. Sargent. Verification and validation of simulation models. *Journal of simulation*, 7(1):12–24, 2013.
34. M. W. Sneddon, J. R. Faeder, and T. Emonet. Efficient modeling, simulation and coarse-graining of biological complexity with NFsim. *Nature Methods*, 8(2):177–183, 2011.
35. J. J. Tyson. Modeling the cell division cycle: cdc2 and cyclin interactions. *Proceedings of the National Academy of Sciences*, 88(16):7328–7332, 1991.
36. T. Warnke, T. Helms, and A. M. Uhrmacher. Syntax and Semantics of a Multi-Level Modeling Language. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, pages 133–144, 2015.
37. R. G. Wiegert. Holism and reductionism in ecology: Hypotheses, scale and systems models. *Oikos*, 53(2):267–269, 1988.

Preprint