Domain-specific Languages for Modeling and Simulation

Dissertation
zur
Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
der Fakultät für Informatik und Elektrotechnik
der Universität Rostock

vorgelegt von

Tom Warnke, geb. am 25.01.1988 in Malchin

aus Rostock

Rostock, 14. September 2020

https://doi.org/10.18453/rosdok_id00002966

Gutachter:
Prof. Dr. Adelinde M. Uhrmacher (Universität Rostock)
Prof. Rocco De Nicola (IMT Lucca)
Prof. Hans Vangheluwe (Universität Antwerpen)

Eingereicht am 14. September 2020
Verteidigt am 8. Januar 2021

Abstract

Simulation models and simulation experiments are increasingly complex. One way to handle this complexity is developing software languages tailored to specific application domains, so-called domain-specific languages (DSLs). This thesis explores the potential of employing DSLs in modeling and simulation. We study different DSL design and implementation techniques and illustrate their benefits for expressing simulation models as well as simulation experiments with several examples.

Regarding simulation models, we focus on discrete-event models based on continuous-time Markov chains (CTMCs). Most of our work revolves around ML-Rules, an rule-based modeling language for biochemical reaction networks. First, we relate the expressive power of ML-Rules to other currently available modeling languages for this application domain. Then we define the abstract syntax and operational semantics for ML-Rules, mapping models to CTMCs in an unambiguous and precise way. Based on the formal definitions, we present two approaches to implement ML-Rules as a DSL. The core of both implementations is finding the matches for the patterns on the left side of ML-Rules' rules. The first approach makes use of ideas from functional programming and realizes ML-Rules as an internal DSL embedded in Scala. The second approach utilizes the language workbench Xtext and object-oriented programming to implement ML-Rules as an external DSL. We relate both approaches to each other and discuss their specific trade-offs, for example regarding computational efficiency. In addition, we demonstrate how DSL implementation techniques can be employed to integrate CTMC-based modeling into Repast Simphony, a software framework for agent-based simulation.

Simulation experiments benefit from DSLs in a different way than simulation models. Here, we focus on more technical issues like repeatability, replicability, reproducibility, and reusability of experiments. The utility of DSLs for expressing simulation experiments is illustrated based on two different DSL implementations. First, we discuss SESSL, an object-oriented Scala-based internal DSL, highlighting diverse ways in which DSLs can support effective simulation experimentation. Based on the ideas of SESSL, we also present a Scala-based internal DSL that is implemented in a purely functional fashion, and discuss the implications.

Zusammenfassung

Simulationsmodelle und -experimente werden immer komplexer. Eine Möglichkeit, dieser Komplexität zu begegnen, ist, auf bestimmte Anwendungsgebiete spezialisierte Softwaresprachen, sogenannte domänenspezifische Sprachen (*DSLs, domain-specific languages*), zu entwickeln. Die vorliegende Arbeit untersucht, wie DSLs in der Modellierung und Simulation eingesetzt werden können. Wir betrachten verschiedene Techniken für Entwicklung und Implementierung von DSLs und illustrieren ihren Nutzen für das Ausdrücken von Simulationsmodellen und -experimenten anhand einiger Beispiele.

In Bezug auf Simulationsmodelle konzentrieren wir uns auf diskret-ereignisbasiert Modelle, die auf Markow-Ketten mit kontinuierlicher Zeitbasis (*CTMCs, continuous-time Markov chains*) basieren. Der größte Teil dieses Abschnitts dreht sich um die regelbasierte Modellierungssprache für biochemische Reaktionsnetze ML-Rules. Als Ausgangspunkt setzen wir die Ausdruckskraft von ML-Rules mit anderen verfügbaren Modellierungssprachen in diesem Anwendungsgebiet in Beziehung. Im Anschluss definieren wir die abstrakte Syntax und formale Semantik von ML-Rules, wodurch eine eindeutige Abbildung von Modellen auf CTMCs hergestellt wird. Auf Grundlage dieser formalen Definitionen stellen wir zwei Ansätze zur Implementierung von ML-Rules in einer DSL vor. Der Kern beider Implementierungen ist das Finden der Vorkommen der Muster auf den linken Regelseiten von ML-Rules. Der erste Ansatz nutzt Ideen aus der funktionalen Programmierung und setzt ML-Rules als interne DSL in Scala um. Der zweite Ansatz nutzt die *language workbench* Xtext und objekt-orientiere Programmierung, um ML-Rules als externe DSL umzusetzen. Wir stellen beide Ansätze nebeneinander und diskutieren die spezifischen Trade-offs, zum Beispiel in Bezug auf Recheneffizienz. Des Weiteren demonstrieren wir, wie CTMC-basierte Modellierung mit Hilfe von DSL-Implementierungstechniken in Repast Simphony, ein Framework für agentenbasierte Simulation, integriert werden kann.

Simulationsexperiment profitieren auf andere Weise als Simulationsmodelle von der Nutzung von DSLs. Hier konzentrieren wir uns auf technischere Aspekte wie Wiederholbarkeit, Replizierbarkeit, Reproduzierbarkeit und Wiederverwendbarkeit von Experimenten. Der Nutzen von DSLs für das Ausdrücken von Simulationsexperimenten wird mit Hilfe zweier DSl-Implementierungen gezeigt. Zunächst behandeln wir SESSL, eine objektorientierte Scala-basierte interne DSL, wobei wir hervorheben, wie die Nutzung von DSLs das effektive Durchführen von Simulationsexperimenten unterstützt. Aufbauend auf den Ideen hinter SESSL stellen wir dann eine Scala-basierte interne DSL vor, die auf rein funktionale Art implementiert ist, und diskutieren die Implikationen.

Acknowledgements

I would have never finished this thesis without the support and assistance of many people.

First and foremost, I want to thank my supervisor Prof. Lin Uhrmacher. With her knowledge and inspiring enthusiasm, she always had the right advice to help me overcome any obstacle on my way to a finished thesis.

I would like to thank my colleagues for their constructive feedback and cooperation. It was invaluable to always have someone to bounce ideas off, and I think much of this thesis is the direct result of the great work atmosphere in the modeling and simulation group. I am grateful to Tobias for introducing me to ML-Rules and to Roland for his work on SESSL.

I want to thank my parents and my sister for their unshakable support. Most importantly, I am grateful to Anna for her patience and for always believing in me.

Contents

1. Introduction

The complexity of simulation models is steadily increasing [50]. One reason is that the growing availability of computational power makes previously infeasible simulations possible. But another driving force are the application domains that spawn increasingly ambitious simulation studies. For example, in 2012 for the first time a "whole-cell" model depicting the functions of all genes in a biological cell was published, made possible by advances in genomics [116]. The model contains and integrates knowledge about a specific bacterium in all its complexity. Therefore, arguably the most valuable result of simulation studies like this is the model itself.

This pattern of creating simulation models as a means to further the understanding of real-world phenomena and processes can be found in different application areas. Simulation has become "a method of first resort" [143]. One prolific domain is cell biology, which has lead to terms such as systems biology, computational biology, or executable cell biology [78]. Here, increasingly complex biological systems are captured in increasingly complex simulation models. Similarly, the social sciences have an interest in "the exploration and understanding of social processes by means of computer simulation", as the Journal of Artificial Societies an Social Simulation (JASSS)[1] puts it [214]. A third example for a scientific domain that employs simulation modeling as a tool for capturing complex phenomena is ecology [93]. In addition to the sciences, many applications in engineering also rely on simulations of increasing complexity [53].

A direct consequence of the complexity of simulation models is that working with these models becomes more complex as well. For example, such models are often stochastic, have many input parameters, or change their dynamics during simulation. To explore the model behavior or make reliable statements about model properties, increasingly complex simulation experiments must be conducted.

In computer science, one central way to address complexity is abstraction [126]. The idea of abstraction is, in fact, the essence of one of the most fundamental concepts in computer science, the lambda calculus. The lambda calculus is a formally defined language with rules for how terms are constructed (syntax) and evaluated (semantics) [189, p. 51ff]. Abstraction is captured by these rules as factoring out elements of a term into a function parameter. This way, unimportant details can be abstracted away and solutions to complex problems can be generalized and expressed succinctly. Thus, the lambda calculus exemplifies how computer science employs languages to express abstractions and handle complexity.

[1]jasss.soc.surrey.ac.uk

1.1. Languages for handling complexity

This thesis applies the idea of using languages that allow to abstract over specific complex application domains in the area of modeling and simulation. Of course, abstraction is the key idea in modeling and simulation, as models are abstractions of real-world phenomena. But languages and abstraction play an important role in modeling and simulation beyond that. We illustrate this with an example.

Assume that we are modeling how the size of a animal population develops. If we assume atomic birth and death events, we could describe the model with statements like the following.

- If the population size is 2, the next population size is either 1 or 3.
- If the population size is 3, the next population size is either 2 or 4.
- If the population size is 4, the next population size is either 3 or 5.

We could continue to enumerate state transitions this way. However, we could also abstract over these statements and make the underlying pattern explicit.

- If the population size is x and $x \geq 2$, the next population size is either $x - 1$ or $x + 1$.

Similarly as in the lambda calculus, we factored out the concrete population size into the variable x, which we then used in arithmetic expressions. By replacing x with concrete numbers and evaluating the arithmetic expressions, we can obtain the statements of the original model description again. In this way, syntax and semantics rules can define languages that allow abstract, precise description of simulation models. Without an upper limit for x, this one sentence even describes an infinite system.

Abstraction through languages is also useful for handling the complexity of simulation experiments. However, as experiments are essentially executable programs, we can rely on standard programming language techniques for abstraction. For example, we can wrap repeated code in a reusable module. As simulation experiments are part of the scientific method, they must support the reliability of experimental results, for example by allowing researchers to communicate, replicate, and adapt experiments. These specific requirements can be handled with appropriate languages, even for complex experiments.

Choosing an appropriate language for expressing simulation models or simulation experiments does not make them less complex. However, choosing an inappropriate language might make them appear more complex than they are. As argued by Fred Brooks in his influential article “No Silver Bullet: Essence and Accidents of Software Engineering”, complexity can be divided into *essential* and *accidental* complexity [37]. Whereas the essential complexity is inherent to the problem at hand (modeling a system or conducting an experiment with a model), the accidental complexity results from using a suboptimal language for solving the problem.

One way to minimize the accidental complexity is developing languages tailored to the problem at hand, so-called *domain-specific languages* (DSLs) [80]. In this thesis we investigate how DSLs can be applied to simulation models and simulation experiments. DSLs for modeling are designed with appropriate domain-specific abstractions, leading to succinct and readable model descriptions. DSLs for experiments focus on supporting the reliability of experimental results.

1.2. Terminology

Before describing the contributions of this thesis, we give some precise definitions of some of the terms we already used informally above. The definitions follow the M&S textbooks by Law [133] and Zeigler, Muzy, and Kofman [255].

We define a *model*[2] as an abstract surrogate of a system, whereas a *simulation* is an execution of a model that, given some input, generates output. It is often desirable to separate model and simulation (algorithm). As Page puts it, "the syntax concomitant with [the simulation algorithm] clutters a programmed model with details that contribute nothing to the description of the behavior of the underlying system. Since ideally, in order to facilitate model analysis, a description free of these and other *implementation* details is preferable, the need for higher level model representational forms becomes evident" [178, p. 21]. The "higher level" model representation often takes the form of a *modeling language* (or *modeling formalism*), which abstracts over models by providing specific syntax and semantics. Modeling languages can be textual or graphical.

The primary use of models is to run *simulation experiments* with them. In his textbook, Cellier describes this relation as follows: "A model (M) for a system (S) and an experiment (E) is anything to which E can be applied to answer a question about S" [45, p. 5]. The idea that a model only represents a system under specific experimental conditions has been formalized as *experimental frame* by Zeigler. Thus, it is important to describe experiments precisely. Such experiment descriptions can be formulated in tailored languages.

1.3. Contributions

This thesis explores the utility of developing domain-specific languages tailored to specific application domains in modeling and simulation. The main contributions are the following:

- We develop the formal syntax and semantics of ML-Rules, an expressive rule-based modeling language for biochemical reaction networks. ML-Rules models are mapped to continuous-time Markov chains (CTMCs), an established formal foundation for discrete-event simulation models.

- We present two different approaches of implementing ML-Rules, as an internal DSL employing functional programming and as an external DSL in the object-oriented programming paradigm. The implementations expose the characteristic trade-offs of both approaches.

[2]Unfortunately, the term "model" is used in many different contexts. Unless otherwise noted, in this thesis the term will mean a simulation model as defined here.

- We integrate the idea of rule-based modeling with CTMC semantics into a framework for agent-based modeling, again employing DSL techniques.
- We present extensions to SESSL, an object-oriented DSL for experiment specification. These extensions highlight diverse ways in which DSLs can support effective simulation experimentation.
- Based on the insights gained during the work on SESSL, we propose expressing simulation experiments as pure functions. This allows us to give guarantees about determinism of complex experiments.

To keep the thesis focused on these contributions, some other related topics will not be covered.

- We focus on stochastic models. Much of this thesis also applies to deterministic simulation, but this distinction will not be considered further.
- We do not discuss graphical languages, but focus on textual languages. Although graphical representations can be richer in terms of transported information, text-based notations are typically more explicit and unambiguous ("The question is not 'Is a picture worth a thousand words?', but 'Does a given picture convey the same thousand words to all viewers?'" [186]).
- There exists a plethora of simulation algorithms for CTMCs. We largely glance over questions of computational efficiency and specific algorithms, and only touch upon simulation algorithms when they interact with the modeling language.

Furthermore, some of the work in this thesis is based on preexisting work. In particular, ML-Rules has been first proposed in 2011 by Maus, Rybacki, and Uhrmacher [149] and continuously developed since then. It has been implemented twice, and we sometimes refer to the most recent implementation authored by Tobias Helms as "ML-Rules 2" [101]. Similarly, SESSL has been first been presented in 2014 [70]. The work described in this thesis is a further development of the initial implementation authored by Roland Ewald.

1.4. Outline

The thesis is structured as follows. In Chapter 2, we give background on software languages and, more specifically, DSLs. The fundamentals of modeling and simulation with CTMCs are given in Chapter 3. Chapter 4 contains an introduction to ML-Rules and related languages. The syntax and semantics of ML-Rules is defined in Chapter 5. Chapter 6 is concerned with the implementation of ML-Rules as well as extending some of its ideas to agent-based modeling. Support for simulation experimentation with DSLs is considered in Chapter 7. Finally, Chapter 8 summarizes the thesis and makes some concluding remarks.

1.5. Bibliographical Note

Some parts of this thesis contain revised content of published articles.

- The formal syntax and semantics of ML-Rules (i.e., parts of Chapter 5) has been published in the following article:

  Tom Warnke, Tobias Helms, and Adelinde M. Uhrmacher. "Syntax and Semantics of a Multi-Level Modeling Language". In: *Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.* New York: ACM, 2015, pp. 133–144. DOI: 10.1145/2769458.2769467

- The integration of a CTMC-based DSL into an agent-based modeling framework (Section 6.6) has been published in the following article, which received the *Best Ph.D. Paper* award at the Winter Simulation Conference 2016:

  Tom Warnke, Oliver Reinhardt, and Adelinde M. Uhrmacher. "Population-based CTMCS and agent-based models". In: *Proceedings of the 2016 Winter Simulation Conference.* IEEE, 2016. DOI: 10.1109/wsc.2016.7822181

- A series of publications describes my work on SESSL, part of which was revised for Section 7.2:

  Tom Warnke, Tobias Helms, and Adelinde M. Uhrmacher. "Reproducible and flexible simulation experiments with ML-Rules and SESSL". in: *Bioinformatics* 34.8 (Nov. 2017). Ed. by Bonnie Berger, pp. 1424–1427. DOI: 10.1093/bioinformatics/btx741

  Tom Warnke and Adelinde M. Uhrmacher. "Complex Simulation Experiments made easy". In: *Proceedings of the 2018 Winter Simulation Conference.* IEEE, 2018. DOI: 10.1109/wsc.2018.8632429

- The idea of using purely functional programming for simulation experiments as presented in Section 7.3 was published in the following article:

  Tom Warnke and Adelinde M. Uhrmacher. "Reproducible parallel simulation experiments via pure functional programming". In: *Proceedings of the 2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT).* IEEE, Oct. 2019. DOI: 10.1109/ds-rt47707.2019.8958655

The following publications I co-authored are also referenced in this thesis:

- An informal introduction to ML-Rules has been published as a book chapter:

  Tobias Helms, Tom Warnke, and Adelinde M. Uhrmacher. "Multi-Level Modeling and Simulation of Cellular Systems: An Introduction to ML-Rules". In: *Modeling Biomolecular Site Dynamics.* Ed. by William S.

Hlavacek. Springer New York, 2019, pp. 141–160. DOI: 10.1007/978-1-4939-9102-0_6

- We shortly touch upon some performance considerations for ML-Rules, which were studied in more depth in the following publications:

  Tobias Helms et al. "Semantics and Efficient Simulation Algorithms of an Expressive Multi-Level Modeling Language". In: *ACM Transactions on Modeling and Computer Simulation* 27.2 (May 2017), 8:1–8:25. DOI: 10.1145/2998499

  Tom Meyer et al. "On Performance Benefits of Code Generation at Runtime for Interpreted Domain-Specific Modeling Languages". In: *Proceedings of the 2018 Winter Simulation Conference*. IEEE, 2018. DOI: 10.1109/WSC.2018.8632545

  Till Köster, Tom Warnke, and Adelinde M. Uhrmacher. "Partial Evaluation via Code Generation for Static Stochastic Reaction Network Models". In: *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. Miami FL Spain: ACM, June 15, 2020, pp. 159–170. ISBN: 978-1-4503-7592-4. DOI: 10.1145/3384441.3395983. (Visited on 07/02/2020)

- The adaptation of continuous-time agent-based modeling as discussed in Section 6.6 has been proposed in the following publications:

  Tom Warnke et al. "ML3: A Language for Compact Modeling of Linked Lives in Computational Demography". In: *Proceedings of the 2015 Winter Simulation Conference*. Piscataway, New Jersey: IEEE, 2015. DOI: 10.1109/wsc.2015.7408382

  Tom Warnke et al. "Modelling and simulating decision processes of linked lives: An approach based on concurrent processes and stochastic race". In: *Population Studies* 71.sup1 (Oct. 2017), pp. 69–83. DOI: 10.1080/00324728.2017.1380960

- Applications of the experiment specification language SESSL have been documented in various settings.

  The idea of analyzing, modifying or generating SESSL experiments (see Section 7.4) was explored in the following articles:

  Danhua Peng et al. "Reusing simulation experiment specifications to support developing models by successive extension". In: *Simulation Modelling Practice and Theory* 68 (Nov. 2016), pp. 33–53. DOI: 10.1016/j.simpat.2016.07.006

  Danhua Peng et al. "Reusing simulation experiment specifications in developing models by successive composition — a case study of the

Wnt/β-catenin signaling pathway". In: *SIMULATION* 93.8 (Apr. 2017), pp. 659–677. DOI: 10.1177/0037549717704314

Andreas Ruscheinski et al. "Generating Simulation Experiments Based on Model Documentations and Templates". In: *Proceedings of the 2018 Winter Simulation Conference*. IEEE, 2018. DOI: 10.1109/wsc.2018.8632515

Andreas Ruscheinski, Tom Warnke, and Adelinde M. Uhrmacher. "Artifact-based Workflows for Supporting Simulation Studies". In: *IEEE Transactions on Knowledge and Data Engineering* (2019), pp. 1–1. DOI: 10.1109/tkde.2019.2899840

The utility of SESSL as a means for facilitating and documenting the calibration and validation of simulation models in a reproducible manner (cf. Section 7.1.2) was discussed in the following articles:

Maria E. Pierce et al. "Developing and validating a multi-level ecological model of eastern Baltic cod (Gadus morhua) in the Bornholm Basin – A case for domain-specific languages". In: *Ecological Modelling* 361 (Oct. 2017), pp. 49–65. DOI: 10.1016/j.ecolmodel.2017.07.012

Oliver Reinhardt et al. "Valid and Reproducible Simulation Studies—Making It Explicit". In: *Simulation Foundations, Methods and Applications*. Ed. by Claus Beisbart and Nicole J. Saam. Springer International Publishing, 2019, pp. 607–627. DOI: 10.1007/978-3-319-70766-2_25

The deployment of SESSL experiments using Maven (Section 7.2.3) was presented in the following paper:

Tom Warnke, Frank Krüger, and Adelinde M. Uhrmacher. "Open Simulation Software - Development and Application". In: *ASIM 2018 - 24. Symposium Simulationstechnik*. ARGESIM Verlag Wien, 2018, pp. 307–312. DOI: 10.11128/arep.56

2. Domain-specific Languages

Before studying domain-specific languages in particular, we consider the terminology for languages in general. We follow Kleppe [120] and adopt the term *software languages* to refer to all artificially designed languages that define the input for some software, including programming languages, specification languages, or (simulation) modeling languages. Similarly, we adopt the term *mogram* to refer to programs, specifications, (simulation) models, or any other artifacts expressed in a software language.

When working with software languages, three fundamental aspects can be distinguished.

Concrete Syntax The concrete syntax describes how the language appears to the user. This typically includes matters like keywords, handling whitespaces, capitalization, or operator preference rules. Generally, describing and processing concrete syntax is tackled with *grammars* and *parsers*, based on formal language theory. Concrete syntax is further discussed in Section 2.2.

Abstract Syntax A mogram can be translated from the concrete syntax to the abstract syntax. Here, the mogram is reduced to its essence and stripped of any superfluous syntactical information. For example, keywords, whitespaces, or the order in which language constructs appear are typically abstracted over. Abstract syntax is often captured in an *abstract syntax tree* (AST) or graph, which can be represented as a *metamodel* in object-oriented programming. These concepts are studied in Section 2.3.

Semantics The third component is the semantics of the language, which assigns a meaning to the mogram. Different semantics can be defined, for example to evaluate the mogram to a result, to translate it to another language, or to typecheck it. The description of the semantics of a language can take diverse forms, ranging from formal, explicit approaches to pragmatic, implicit ones. Section 2.4 considers semantics in more detail.

The pipeline consisting of these components is visualized in Figure 2.1. Sometimes, more intermediate stages are defined, but for the purpose of this thesis these three language aspects suffice. Most importantly, this framework allows characterizing the approaches that are introduced in the remainder of this chapter. Before looking further into these approaches, we investigate the nature of domain-specific languages.

2.1. General-purpose and domain-specific languages

Software languages can be distinguished regarding their genericity. Most mainstream programming languages, for example C, C++, or Java, have been designed to be applicable

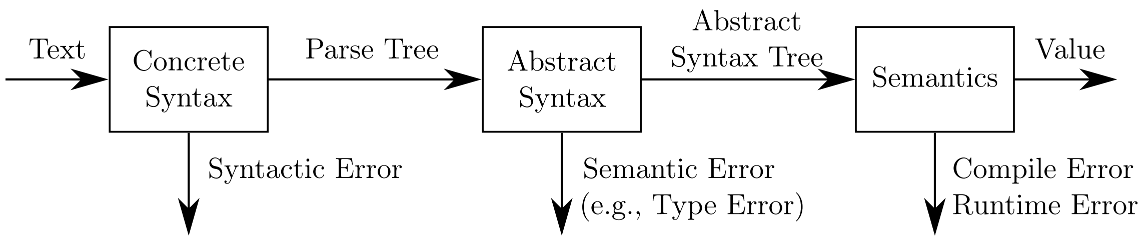


Figure 2.1.: The main components of a language definition at work. By passing through these components, the set of possible inputs is constrained more and more. Processing input texts according to the concrete syntax produces a parse tree for syntactically correct inputs. The parse tree is then further processed to an abstract syntax tree, making sure that the input is free of semantic errors. The semantics is then only invoked on valid abstract syntax trees and evaluates them to a result value or encounters an error during evaluation.

in diverse application domains. They have been used to build server and desktop client software, mobile applications, or games. A single programming language that supports such diverse application areas is a *general-purpose (programming) language* (GPL).

In contrast to a GPL, a *domain-specific language* (DSL) is not designed for developing a complete software system, but rather for developing an individual aspect of such a system. In his book “Domain-specific languages”, Martin Fowler defines a DSL as a programming language that is tailored to a certain application domain, but less expressive than a GPL [80, p. 27f]. DSLs are typically not Turing-complete and, for example, do often not include loops, conditional branching, or recursive functions, which limits DSLs to solving specific problems[1]. Thus, in the sense of formal expressiveness, DSLs are generally less expressive than GPLs.

But formal expressiveness is not the only, arguably not even the most important measure for a language, as stated by Alan Perlis: “Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy” [185]. Instead, *practical expressiveness* measures “how readily ideas can be expressed” in a language, relying, for example, on the language’s conciseness and readability [74, 75]. Thus, in the sense of practical expressiveness, DSLs are generally more expressive than GPLs.

Alternatively, an article by van Deursen et al. characterizes DSLs as programming languages that offer “expressive power focused on, and usually restricted to, a particular problem domain.” [61] This definition also emphasizes that, in comparison to GPLs, DSLs are *more* expressive in their targeted application domain, but *less* expressive in the sense of being generally applicable. This matches the above notions of practical and formal expressiveness, respectively.

Depending on the specific problem domain, the practical expressiveness of a DSL can

[1]There are examples of DSLs that are Turing-complete, but not necessarily deliberately. For example, the input language for the UNIX typesetting formatter `troff` “has conditionals and recursion but not loops; it is accidentally Turing-complete” [195, p. 195].

depend on quite different properties. For example, SQL is a DSL that allows for querying databases by providing operators such as `SELECT`, `FROM`, and `WHERE`. SQL queries are typically more concise and readable than equivalent code in a GPL. Moreover, in comparison to GPL code, SQL queries have a simple declarative structure, which facilitates analyzing and optimizing queries rather than just executing them.

The declarative nature of many DSLs, as exemplified by SQL, means that “they can be viewed as specification languages, as well as programming languages” [61]. Thus, DSLs can not only be used to replace executable code (as SQL does, for example), but also to encode the structure and layout of a document (e.g., LaTeX or Markdown), or to specify a mathematical object (e.g., a system of differential equations or a stochastic process). As these examples show, the declarativity of DSLs also facilitates integrating knowledge, metaphors, and semantics of the domain.

Independently of its genericity, a software language comprises concrete syntax, abstract syntax, and semantics. Many language implementation techniques can be (and have been) used for GPLs as well as for DSLs. Consequently, unless otherwise noted, the following description apply to GPLs as well as DSLs.

2.2. Concrete syntax

The concrete syntax of a language describes the mograms that belong to the language. The main task regarding concrete syntax in the context of textual software languages is translating valid input strings into a tree that describes the syntactical structure. This process of deciding whether a mogram is valid and, if so, translating it, is called *parsing* [3, p. 30]. The resulting *parse tree* is then typically translated further to a representation of the mogram’s abstract syntax (see Section 2.3).

We begin our study of concrete syntax with formal language theory, which sets (and also continues to push) the limits of parsable concrete syntax. The central result of these theoretical considerations is that designers of software languages are largely limited to the language class of *context-free languages*. Based on this result, we subsequently look into *parser generators* and *parser combinators*, which are two approaches that support the development of parsers for context-free languages.

2.2.1. Formal language theory

Formal language theory provides some insights regarding the construction of languages with grammars as well as the inverse operation, parsing a text and interpreting it as part of the language [161, p. 9ff.].

Formally, the terms *language* and *grammar* can be defined as follows: A language L is a subset of all strings of finite length over a finite alphabet X. The strings that belong to L are called *sentences*. Whereas languages are potentially infinite sets, they can be finitely described with a grammar that is able to generate all its sentences. A grammar $G = (X, V, S, P)$ contains four elements: the alphabet of *terminal symbols* X (from which the sentences of the language are built), the alphabet of *nonterminal symbols* V (which are placeholders that do not appear in final sentences), a start symbol $S \in V$, and the set of *productions* P. Each production is a rule $u \rightarrow v$, stating that the string u can be replaced by the string v. u needs to contain at least one nonterminal symbol. The language $L(G)$ is defined by the grammar G as the reflexive transitive closure of applying the productions to the start symbol until all nonterminal symbols have been replaced.

The Chomsky hierarchy describes how different classes of formal languages relate to each other, based on properties of the productions [49]. It states that right-linear languages are a proper subset of context-free languages, context-free languages are a proper subset of context-sensitive languages, and context-sensitive languages are a proper subset of recursive enumerable sets of strings. Each of these language types can be generated with a grammar with specific restrictions. For example, right-linear languages are generated if the grammar only contains productions of the form $A \rightarrow aB$ or $A \rightarrow a$, where A and B are nonterminal symbols and a is a terminal symbol. Right-linear languages can equivalently be generated with regular expressions, which is why they are sometimes called regular languages. Context-free languages are generated by grammars with productions of the form $A \rightarrow \omega$, where A is a nonterminal symbol and ω is a non-empty string that may consist of terminal and nonterminal symbols. Context-sensitive languages then allow that the nonterminal symbol on the left side is embedded in some context, and productions have the form $\phi_1 A \phi_2 \rightarrow \phi_1 \omega \phi_2$, where A is a nonterminal symbol and ω, ϕ_1, and ϕ_2 are strings that may consist of terminal and nonterminal symbols. Grammars that are not restricted generate recursive enumerable sets of strings.

Based on this hierarchy, the expressive power of the different language classes can be characterized. For example, the language $\{a^n b^n\}$ can be shown to be context-free, whereas the language $\{a^n b^n c^n\}$ can be shown to be context-sensitive, but not context-free [161, p. 12].

Similarly to the relation between language classes and grammar restrictions, the different language classes require specific parsing techniques. Regular languages can be parsed with finite automata, context-free languages can be parsed with push-down automata. Novel parsing techniques, for example for context-sensitive languages, are still actively researched [132]. This theoretical classification has practical consequences. Most importantly, most programming languages are context-free languages, as this class is easy enough to parse and powerful enough to express typical programming language constructs [161, p. 14]. For example, context-free languages can express that scopes (e.g., defined with `{ }`) must be nested correctly.

Due to the structure of the production rules in context-free grammars, the derivation of a sentence can be shown in a derivation tree. The start symbol is the root of the tree, and each nonterminal symbol has as its children the symbols that replaced it. Nonterminal symbols are the leafs of the tree. This tree structure is also the basis for parsing. Whereas the derivation tree defines the produced sentence, parsing a sentence results in a parse tree.

Two main approaches exist to parse context-free languages: top-down parsing and bottom-up parsing. They impose further restrictions on the formulation of the grammar: top-down parsing is related to the grammar class LL, and bottom-up parsing is related to the grammar class LR. Other, more general prerequisites for parsing exist as well. For example, an important property of grammars is that they are *unambiguous*, that is, for every given input there is at most one way to parse it (i.e., at most one parse tree). A notable counterexample to unambiguity is the so-called "dangling-else": the expression `if B1 then if B2 then S1 else S2` can be parsed as `if B1 then (if B2 then S1) else S2` or as `if B1 then (if B2 then S1 else S2)` [1].

$LL(k)$ grammars are grammars in which the possible derivations of each nonterminal have distinct prefixes of a length $< k$. In other words, when parsing it suffices to look at the next k tokens to decide which derivation produced the input. This allows for writing efficient top-down parsers that operate without backtracking. However, it also imposes some constraints on how the grammar can be constructed, and also on what languages can be described with such a grammar. For example, $LL(k)$ grammars are not allowed to be left-recursive.

$LR(k)$ grammars can describe more languages than $LL(k)$ grammars and can be used for efficient bottom-up parsing. In a $LR(k)$ grammar, every sentence can be produced with a rightmost derivation, that is a derivation that replaces the rightmost nonterminal symbol. To parse such a sentence, this derivation can be reversed, which is possible without backtracking if the grammar is unambiguous. The according parsing technique is shift-reduce-parsing.

Formal language theory gives some hints about the limits for the communication between humans and computers. For example, the classification of languages regarding their parsability influences the design of programming languages. In particular, it shows that properties such as unambiguity are important prerequisites for processing a language with a computer. This also implies that natural languages such as English are hard to parse: they typically allow ambiguous sentences such as "I saw a man on a hill with a telescope"[2]. Consequently, the freedom in the design of software languages (including DSLs) is not unlimited—the syntax of the language must still allow for efficient parsing, according to the considerations above.

2.2.2. Parser generators

Although parsers for context-free languages can be implemented by hand, the implementation of a parser can be facilitated by a parser generator. The first tools for parser generation were developed as early as the 1970s, when `lex` and `yacc` were released. Their more modern GNU counterparts `flex` and `bison` were developed in the 1980s [138, p. 9]. Both tool pairs can generate C code that implements a parser. Whereas `lex` tokenizes an input text based on regular expressions (lexical analysis), `yacc` parses the tokes stream in a syntax specified by a context-free grammar (syntactical analysis). `yacc` processes $LALR$ (an LR subclass) grammars specified in simplified Backus-Naur-Form (BNF). A grammar rule in `yacc` can be annotated with code to execute after successfully processing the rule. Because the code can access the already parsed elements of the rule, the parser can perform arbitrary actions with the recognized input. One option is directly evaluating the input and returning some result, for example when parsing constant arithmetic expressions. The following `yacc` grammar snippet defines production rules for the nonterminal symbol

[2] https://www.byrdseed.com/ambiguous-sentences/

`exp`, using the terminal symbols `NUMBER` and `ADD` [138, p. 11]. It allows constructing addition expressions such as `3 + 4 + 5`. The C code in brackets is executed after parsing a rule and adds the numerical value of the right side to the result of recursively evaluating the left side. The sum is assigned to the node representing the addition, thereby evaluating the addition "on the fly".

```
exp: NUMBER
   | exp ADD NUMBER { $$ = $1 + $3; }
   ;
```

Note that the grammar snippet is left-recursive: the leftmost symbol in one rule for `exp` is `exp` itself. Left-recursion would be problematic for a top-down (LL) parser, but the bottom-up parser generated by `yacc` can handle left-recursion without a problem.

The ideas of `lex` and `yacc` have also been implemented in more modern parser generators. Most notable, ANTLR is a heavily used parser generator, particularly in the Java ecosystem [180]. ANTLR v3 employs code annotations for grammar rules similarly as `yacc`. Whereas ANTLR v3 uses LL parsing, the more recent ANTLR v4 uses Adaptive LL (ALL) parsing, which includes parsing-time checks. ANTLR v4 also separates the grammar definition from the code that is invoked for individual rules, and relies on the listener and visitor patterns to decouple the code from the grammar. This facilitates using a grammar with different target languages. The listener approach allows for implementing methods that are invoked before or after a grammar rule has been processed, facilitating an approach similar to the code annotations in `yacc`. Results from evaluating individual tree nodes can be stored in an auxiliary data structure and retrieved when necessary. On the other hand, the visitor approach allows for controlling how the AST is traversed and returning a value for each processed tree node. This facilitates processing the AST with side-effect-free functions.

As stated in the previous section, the language-theoretical foundations limit the generality of context-free grammars and, consequently, generated parsers. Parser generators are most useful for quickly implementing simple languages. While generating a parser from a grammar, they make sure that the grammar satisfies the prerequisites for parsing (such as unambiguity). In compiler suites for GPLs, however, the additional effort for implementing a hand-written parser is often justified by the flexibility of the implementation. For example, `GCC` and `Clang` rely on hand-written parsers, citing as reasons the easier integration of diagnostics and error reporting [52, 166].

2.2.3. Parser combinators

As an alternative to defining grammars and generating parser code, the implementation of parsers can also be supported by parser combinators. Parser combinators are a concept that was developed in the area of functional programming [234]. A parser can be represented as a function that maps some input to a list of pairs of a parsing result and the remaining input (a list of results because several successes are possible if the grammar is ambiguous). With functions being first-class values, functional programming languages enable combinators that allow combining individual parsers to more complex parsers.

The resulting parsing technique is recursive-descent parsing, which is a top-down parsing technique [3, p. 181]. Consequently, it works best with grammars of the $LL(1)$ class, and requires inefficient backtracking for other grammars [171, p. 754]. Although many languages can be described with an $LL(1)$ grammar, this again imposes constraints on what languages can be parsed and how the grammars have to be formulated. For example, left-recursion needs to be eliminated for using an LL parser, which often makes the resulting parse trees harder to process [3, p. 49].

The big advantage of parser combinators is that they can be realized as simple libraries, relying solely on the features that functional programming languages offer. By defining the parser as a function, functional composition can be used to combine parsers. For example, alternative productions can be realized by concatenating the lists that result from trying both productions on the same input. In Haskell, this can be expressed as follows [108]:

```haskell
newtype Parser a = Parser (String -> [(a, String)])

plus :: Parser a -> Parser a -> Parser a
(Parser p) `plus` (Parser q) = Parser (\inp -> (p inp) ++ (q inp))
```

Using monadic function composition, parsers can be sequenced, for example to parse a repetition of a given parser [109]. Haskell's `do`-notation provides a succinct syntax for this:

```haskell
many :: Parser a -> Parser [a]
many p = neSeq `plus` return []
  where neSeq = do a <- p
                   as <- many p
                   return (a:as)
```

Based on some primitive parsers and some simple combinators, parsers can be composed in arbitrarily complex ways in idiomatic functional programs. Thus, parser combinators require significantly less technical overhead than parser generators, as no grammar has to be written as input to some code generation component. However, this also means that parsers defined via combinators are not checked for properties such as ambiguity. It is very easy to create a nonterminating left-recursive parser with parser combinators. In addition, parser combinators typically operate directly on textual input instead of token streams created by lexical analysis. Consequently, this abstraction, which, for example, can remove whitespaces, must be integrated into the parser.

2.2.4. Fluent interfaces

In contrast to the theoretically grounded parser approaches, fluent interfaces are a simple, pragmatic way to define a language [80, p. 343ff.]. The idea here is to define methods that return objects that again provide appropriate methods. When method calls are chained, the resulting code is, given appropriately named methods, readable.

Fluent interfaces are typically used to implement DSLs in small, well-defined application domains. They also require an existing object-oriented GPL that provides a system of classes and objects. For example, the library `AssertJ` uses a fluent style to define assertions for Java unit tests [8]. `AssertJ` allows formulating assertions as the following:

```
assertThat(fellowshipOfTheRing).hasSize(9)
                               .contains(frodo, sam)
                               .doesNotContain(sauron);
```

Another example for fluent interfaces is the variant of the Builder pattern that Joshua Bloch suggests for constructing objects [27, p. 10ff.]. Finally, the Stream API introduced in Java 8 supports expressing stream pipelines with a fluent interface [112].

2.3. Abstract syntax

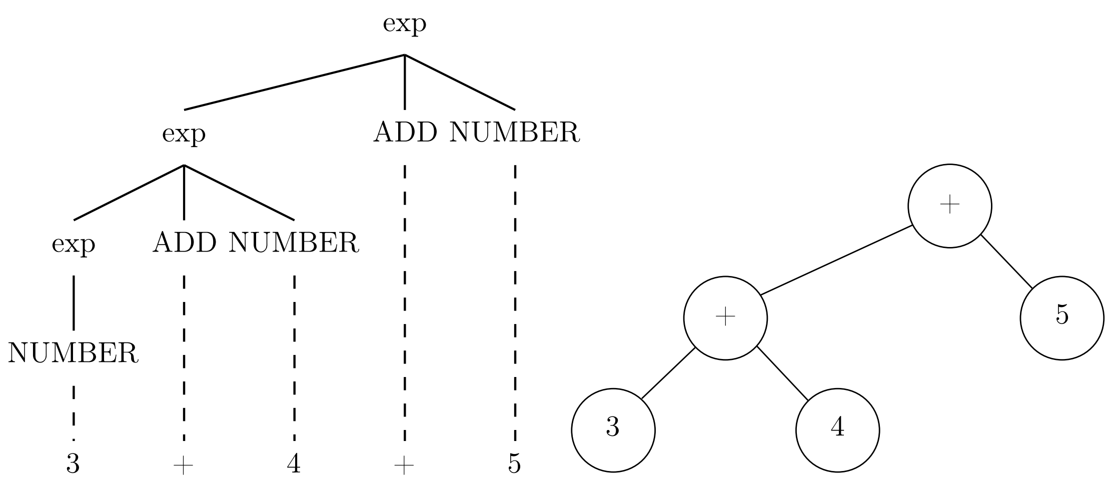
After a mogram has been parsed, the relevant information is extracted and superfluous information discarded. The structure of the resulting information is captured by the abstract syntax. It is abstract as it does not include, for example, where in an expression parentheses were used; instead, the relation of operators and operands is directly represented. Nevertheless, the abstract syntax can be defined with BNF production rules similarly to the concrete syntax. Technically, the abstract syntax often takes the form of a tree, the abstract syntax tree (AST). After shortly studying ASTs in more detail, we look into metamodels, an object-oriented flavor of abstract syntax representation, and into representing abstract syntax in functional languages.

2.3.1. Abstract syntax trees and abstract syntax graphs

Whereas a parse tree is tightly connected to the concrete syntax, an AST allows decoupling parsing a language from further processing it (see Figure 2.2). Many checks that are hard or impossible during parsing can be easily evaluated on the AST [3, p. 287]. For example, many languages allow referring to an identifier that is declared later in the mogram. Parsing can not resolve such forward references, at least not in one pass. The AST can abstract over order of definitions, which removes the “forward” in forward references.

When implementing a DSL, the representation of the abstract syntax is sometimes called a *semantic model* [80, p. 159ff.]. As DSLs are created for specific domains, the semantic model can neglect the concrete syntax even more and instead adopt domain concepts. In traditional compilers, a similar level of abstraction is sometimes used in the intermediate representation that separates front end and back end [3, p. 463].

Representing references inside a mogram, for example to an identifier, violates the tree property of the AST. Instead, the abstract syntax is then captured in a directed acyclic graph (with a spanning tree) [3, p. 290f.]. Besides references to identifiers, non-tree edges in this graph can result from recursive definitions or sharing sub-expressions [173]. Decorating the AST with additional edges makes subsequent analyses and translations much easier (see Section 2.4). We now study one way to construct rich abstract syntax models.



(a) The parse tree.

(b) The abstract syntax tree.

Figure 2.2.: The parse tree and the corresponding abstract syntax tree for the expression $3 + 4 + 5$. Whereas the parse tree strictly follows the grammar rules, the nodes in the abstract syntax tree are operators and operands, encoding the logical structure of the expression.

2.3.2. Metamodels and language workbenches

In object-oriented programming, the unified modeling language (UML) is a generic framework for modeling software. As a less generic (and therefor easier to handle) alternative, DSLs for modeling software are increasingly popular in the field of domain-specific modeling/model-driven engineering [81]. The rise of DSLs for modeling software has raised the challenge of providing tools for each new DSLs [82]. *Language workbenches* have been developed to address this challenge by supporting the rapid creation of editor, analyzer, or code generation components for new languages [69]. This is comparable to parser generators, which simplify the arduous process of developing tools for lexical and syntactical analysis.

The pivotal point of using language workbenches is the generation of a *metamodel*. For a given language, the metamodel describes the types of nodes in the abstract syntax tree/graph as well as their relation in an object-oriented fashion [82]. For example, each node type is represented by a class or interface, and an edge between nodes is an association between classes. Such a metamodel can be captured in languages for describing metamodels, such as the UML (which, in some sense, makes the UML a metametamodel!). Figure 2.3 shows an exemplary metamodel for a simple addition language.

The inference of better (i.e., more precise, expressive, etc.) metamodels is the driving force of language workbench development. Based on the metamodel, additional features such as generation of editors, code generation, but also combining and reusing language definitions are provided. In the following, we shortly look into some examples for language workbenches.

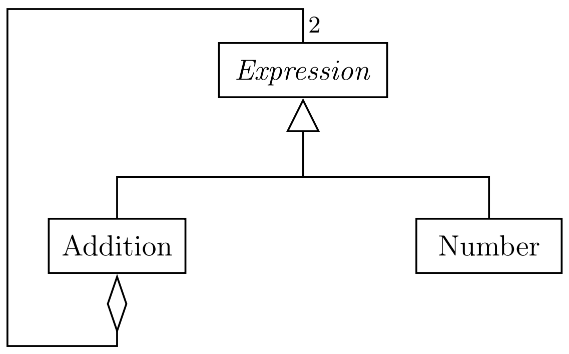


Figure 2.3.: A UML class diagram describing a simple metamodel for a language of nested addition expressions. Addition and Number are subclasses of the abstract class Expression. An Addition contains two Expression instances.

Xtext is a language workbench that is developed as part of the Eclipse ecosystem [63, 254]. It relies on ANTLR 3 for parser generation and, for a given grammar, automatically generates an Eclipse plug-in that allows text editing with syntax highlighting, auto-completion, jump-to-definition, and many other features known form modern programming language IDEs. The inferred metamodel is based on the Eclipse Modeling Framework (EMF) [21] and exploits annotations of the grammar for the concrete syntax. For a metamodel, Xtext generates Java code, providing, for example, stub files for implementing scoping, cross-references, and static analysis and validation. Moreover, Xtext enables serializing parsed DSL mograms back to their DSL representation ("unparsing").

MontiCore is based on Eclipse, but in contrast to Xtext, it originated in academic research [125]. MontiCore focuses on defining DSLs in a modular fashion to enable reuse and composition. From the description of the concrete syntax (given as a grammar), MontiCore infers a metamodel based on annotations of the grammar. To this end, various object-oriented extensions are integrated into the grammar specification: an inheritance relationship between nonterminals, interfaces for nonterminals, and abstract nonterminals, as well as associations between nonterminals. This extended grammar is then transformed to a standard ANTLR EBNF grammar, and the additional information is exploited for metamodel inference. This way, more sophisticated class structures can be generated than with a pure ANTLR grammar. Grammars augmented this way can then be composed through inheritance, where a new grammar inherits the production rules of a number of existing grammars, and embedding, where a nonterminal in the host grammar is resolved by rules of an embedded grammar. Again, MontiCore infers the metamodel for composed grammars automatically. This facilitates the independent development of individual DSLs. As typical for language workbenches, MontiCore supports the generation of further DSL tools such as an Eclipse editor plug-in.

AToM^3 and its successor AToMPM are language workbenches for multi-formalism metamodeling [131, 225]. Here, metamodels for different formalisms can be specified and transformations between those metamodels can be defined. As these transformations operate on the abstract syntax graphs of the formalisms, they can all be expressed with graph grammars. This way, diverse components of a complex system can be modeled in the most appropriate formalism, and to simulate the overall system all components are transformed to a universal formalism, for example DEVS.

A language workbench that follows another approach is Jetbrains' Meta Programming System (MPS) [181, 162]. MPS does not rely on pure textual DSL programs, but rather allows projectional editing of the AST. Because the abstract syntax does not have to be inferred from some textual concrete syntax, no grammar or parser is required. Instead, DSLs are defined as a structure of concepts of which the AST is constructed. Consequently, DSLs can integrate non-parseable syntax such as diagrams, tables, or mathematical notations. For these syntactical forms, different editor components can be created, reused, and combined. As a result, an IDE with the usual features can be generated for a DSL. Similar to Xtext and MontiCore, the primary target of MPS DSLs is the generation of Java code.

2.3.3. Deep and shallow embeddings in functional languages

The functional programming community has developed a particular understanding of embedded languages. Because embedding a language requires an existing GPL, this is typically used to implement a DSL on top of a general-purpose functional programming language.

Inspired by FP's mathematical foundations, the vocabulary of a DSL is often equated to an *algebra* [92]. Here, an Ω-algebra is defined via a set of operator symbols Ω, each $\omega \in \Omega$ having an arity $ar(\omega)$ [188]. The relations of the operator symbols from Ω is described by a set E of equations, constructed with the help of variables. An Ω-algebra A is then a set $|A|$ and an interpretation function a_ω that maps each operator symbol $\omega \in \Omega$ to a function $|A|^{ar(\omega)} \rightarrow |A|$ in such a way that the equations in E are satisfied when replacing the variables with values from $|A|$.

Whereas the concrete syntax is largely determined by the host language, the algebra takes the role of the abstract syntax. By defining a DSL as an algebra, the interpretation function formalizes the separation of the abstract syntax from its semantics. Two major approaches to implement algebraic vocabularies in functional programming languages have been proposed: the *deep* and the *shallow* embedding [86]. The deep embedding reifies the operator symbols as constructors of an algebraic data type (ADT). Thus, a DSL term is represented by an AST. The shallow embedding, on the other hand, directly evaluates the operators to values of the host language.

Gibbons and Wu give the classical example to illustrate the difference between deep and shallow embeddings and its implications [86, 222]. Consider a very simple language of integer literals and addition, allowing expressions such as the following:

```
eval(add(add(lit(3), lit(4)), lit(5)))
```

`lit` (integer `literal`) and `add` (integer `addition`) are the operator symbols of a very simple algebra with arities one and two, respectively. We want to provide an interpretation of the algebra in the set of integers.

A deep embedding of the language includes a recursive ADT, to which expressions are converted. The resulting tree of ADT values corresponds to the AST and can be evaluated with a recursive function (more precisely, a fold):

```
sealed trait Expr
case class Lit(a: Int) extends Expr
case class Add(a: Expr, b: Expr) extends Expr

def lit(a: Int): Expr = Lit(a)

def add(a: Expr, b: Expr): Expr = Add(a, b)

def eval(e: Expr): Int = e match {
  case Lit(a) => a
  case Add(a, b) => eval(a) + eval(b)
}
```

A shallow embedding of the language directly applies the semantics of the evaluation to each subexpression. Consequently, the eventual `eval` is reduced to the identity function:

```
def lit(a: Int): Int = a

def add(a: Int, b: Int): Int = a + b

def eval(e: Int): Int = e
```

When comparing the extensibility of both approaches, the *expression problem* can be observed [235]: The deep embedding can be easily extended with additional evaluation functions (e.g., to generate a string representation of the expression), but adding new constructs to the language (e.g., multiplication) requires changing all evaluators. Conversely, the shallow embedding makes adding more language constructs easy, but adding new evaluators requires revisiting all language constructs.

Finding ways to mitigate the expression problem is the subject of recent and current programming language research (termed "the holy grail in embedded language implementation" [222, p. 24]). Although these approaches are not the focus of this thesis, some important examples need to be mentioned for the sake of completeness.

- It has been shown that *free monads* (an example of a deep embedding) can be combined without running into the expression problem [224].
- An approach to solve the expression problem that is based on shallow embedding is the *tagless final encoding* [44].
- Deep and shallow embeddings can also be combined to tackle the expression problem [222].

In general, DSLs embedded in functional programming languages often benefit from the powerful type checkers of their host language (e.g., Haskell, OCaml, or Scala). This way, many classes of errors in the DSL code can be detected with minimal implementation effort by the DSL developer.

The main downsides of deeply embedded DSLs is that their implementation often requires replicating elements of the host language's compiler [66]. In shallow embeddings, on the other hand, opportunities for static analysis are severely limited.

2.4. Semantics

The formalizations of the concrete and abstract syntax of software languages (grammars, ASTs, etc.) have largely been adopted by practitioners. However, the semantics of software languages, describing the meaning of a mogram in the language, is often not formalized in practice. In the seminal compiler text book by Aho, Sethi, and Ullman ("the dragon book"), the authors wrote in 1985 [3, p. 25]:

> With the notations currently available, the semantics of a language is much more difficult to describe than the syntax. Consequently, for specifying the semantics of a language we shall use informal descriptions and suggestive examples.

It seems that in the 35 years since these sentences were written, this has not changed significantly. To date, few programming languages have a complete formal specification of semantics, with Standard ML being a notable example [219, 160]. Many practical programming languages are designed with an informally specified semantics. "A common form is a reference manual, which is usually a careful narrative description of the meaning of each construction in the language." [160, p. ix]. Sometimes, the languages are retrofitted with formal semantics, at least for parts of the language. For example, formal semantics for Java 1.4 and a subset of C 99 have recently been proposed [28, 67].

Informal as well as formal semantics are defined on the abstract syntax of the language, often by recursively traversing the AST and using non-tree edges to resolve references etc. It is also important to note that informal and formal semantics are not mutually exclusive. For example, by providing a formal type system and checking types of an input program, run-time type errors in (informal) implementations can be avoided [189, p. 4]. Moreover, formal semantics allows analyzing and transforming programs while being "correct-by-construction" [153]. Similarly, semantic information facilitates optimizations of a program, for example to determine whether variables can be safely inlined [3, pp. 14f].

In the following, we consider informal specifications of semantics in the form of reference algorithms for interpretation or compilation[3]. Afterwards, we look into formally specified semantics.

[3]DEVS is an example for a simulation modeling language a semantics that is defined by a reference algorithm. Zeigler, Muzy, and Kofman [255, p. 196] give the semantics of DEVS as "abstract simulators" that "characterize what has to be done to execute [. . .] models".

2.4.1. Interpretation and Compilation

Compilers and interpreters are algorithms that consume some abstract representation of a mogram (such as an AST). A compiler translates the mogram into another language, whereas an interpreter evaluates it [3, p. 1ff.]. As such, compilers and interpreters informally define the semantics of a language through a reference implementation of the language semantics. This "pragmatic" style of describing the semantics of a language is more common than the formal one discussed in the next section [120].

A way to relate interpretation and compilation is Futamura's work regarding partial evaluation [84]. The fundamental difference between both approaches is that a compilation algorithm distinguishes compile-time and run-time, whereas an interpreter only knows run-time. This means that (typically) an interpreter is easier to implement, but interpreted execution is less efficient than executing a compiled program.

Futamura's ideas revolve around a transformation of computation processes. By providing values for a subset of the inputs of a computation, a *specialized*, possibly simpler computation can be obtained. For example, consider the following computation:

$$f(x, y) = x \times (x \times x + x + y + 1) + y \times y$$

If we evaluate this for $x = 1$ and $y = 1, 2, \ldots, n$, we need to perform $3n$ multiplications and $4n$ additions. Compare this to the following computation where $x = 1$ has already been evaluated (which requires 2 additions and 2 multiplications):

$$f(1, y) = f_{x=1}(y) = 1 \times (1 \times 1 + 1 + y + 1) + y \times y = 3 + y + y \times y$$

Now evaluating for $y = 1, 2, \ldots, n$ requires n multiplications and $2n$ additions. To get from $f(x, y)$ to $f_{x=1}(y)$ we partially evaluate f with $x = 1$. More generally, we transform a computation $\pi(c_1, \ldots, c_n, r_1, \ldots, r_m)$ by already assigning values $c'_1, \ldots, c'_n$ to the parameters $c_1, \ldots, c_n$, yielding a computation with the remaining parameters $r_1, \ldots, r_m$. When assigning values $r'_1, \ldots, r'_m$ to the parameters $r_1, \ldots, r_m$, the transformed computation yield the same results as the original one. This transformation is called α.

$$\pi(c'_1, \ldots, c'_n, r'_1, \ldots, r'_m) = \alpha(\pi, c'_1, \ldots, c'_n)(r'_1, \ldots, r'_m)$$

The Futamura projections can now be defined based on an interpreter $\pi(p, i_1, \ldots, i_n)$ for a program p and inputs $i_1, \ldots, i_n$. π takes as inputs the program (source code) itself and the inputs for the program.

$$\begin{aligned} & \pi(p, i_1, \ldots, i_n) && \text{Interpreter} \\ = {} & \alpha(\pi, p)(i_1, \ldots, i_n) && \text{Futamura Projection 1: Executable} \\ = {} & \alpha(\alpha, \pi)(p)(i_1, \ldots, i_n) && \text{Futamura Projection 2: Compiler} \\ = {} & \alpha(\alpha, \alpha)(\pi)(p)(i_1, \ldots, i_n) && \text{Futamura Projection 3: Compiler-Compiler} \end{aligned}$$

The first Futamura Projection results from specializing the interpreter to the program, yielding the executable program $\alpha(\pi, p)$. Here, parsing, type-checking, etc. are done and the source code p is not needed anymore. Next, by another level of specializing, we can abstract over the program p as well. The result is a compiler: a computation $\alpha(\alpha, \pi)$ that computes an executable from each given program code p. Adding another level of specialization, the computation abstracts over the interpreter and, thus, yields a computation that transforms any interpreter to a compiler. In practice, these ideas have mostly been applied in constrained contexts, for example in the form of multi-stage programming [198].

This shows that interpretation and compilation are two points in the continuum of abstracting and specializing the execution of programs. In many modern programming ecosystems, however, the concept of Virtual Machines (VMs) introduces another level of abstraction. For example, programs written in Java, Scala, or Groovy can be *compiled* to bytecode, which is then *interpreted* by the Java Virtual Machine (JVM). Languages such as Erlang or Ruby use VMs as their primary execution platform in a similar way.

2.4.2. Type systems and static analysis

Type systems play a central role in programming language research. For example, the influential textbook by Benjamin C. Pierce is termed “Types and programming languages” [189]. In that book, a type system is defined as “a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute” [189, p. 1]. In particular, using a type system to determine the type of a term does not require the evaluation of that term. Thus, with respect to Figure 2.1 type checking is an additional method to detect errors early before evaluating a mogram. All operations on a mogram that occur after type checking can then rely on the types of the mogram being correct. For example, it is claimed that in Haskell, a programming language with a powerful type system, a program “usually works” once it passes the type checker [249]. This can make a significant difference for the programming experience.

Formally, a type system is defined via a ternary *typing relation* $\Gamma \vdash t : T$, where $\Gamma = t_1 : T_1, t_2 : T_2, \ldots$ is a *typing context*. Then, $\Gamma \vdash t : T$ means that in the context Γ the term t has the type T. The context can be omitted if it is empty. For example, the typing rules for a `let-in` expression can be expressed as follows [189, p. 124]:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2}$$

T_2 is the overall type of the expression `let` $x = t_1$ `in` t_2 in the context Γ, given that the type of t_1 can be determined to be T_1 in the context Γ and, in the context Γ extended with the information that x is of type T_1, it can be determined that t_2 is of type T_2.

In some programming languages, terms are explicitly annotated with type information (e.g., Java or C). Other programming languages, in particular functional languages such as ML or Haskell, are able to infer types and do not require type annotations. Programming languages can also be distinguished regarding the time at which type checking is done. Statically typed languages check types at compile-time, whereas dynamically typed languages check types at run-time [189, p. 2]. Thus, static typing is more useful for catching errors early and avoiding a specific classes of run-time errors, whereas dynamic typing enables more rapid development without needing to satisfy the type checker. Whereas this reasoning applies mostly to GPLs, it is also valid for DSLs.

Static type checking is an example for *static analysis*, that is the inference of properties of a mogram without executing it. Other well-known static analyses for GPLs include escape analysis [24], which statically determines the lifetime of data to support memory management, and Rust's borrow checker [148], which detects possible memory errors. However, static analysis has also been successfully applied to DSLs. For example, the simulation modeling language Kappa is equipped with a static analyzer that allows to statically determine unreachable states [33]. As DSLs are generally simpler than GPLs, static analysis should be simpler for DSLs as well.

2.4.3. Formal semantics

Alternatively to providing a reference implementation of a compiler or interpreter for a software language, semantics can also be defined formally. The former approach *implicitly* fixes the semantics of a language, constraining all future implementations to yield the same results as the reference implementation. A formal definition of the semantics is *explicit*, specifying the behavior of all language implementations declaratively. Typically, this means that the semantics assign a meaning to each construct that appears in the AST.

In general, it should be more feasible to define formal semantics for a DSL than a GPL. DSLs are (by definition) less powerful in terms of theoretical expressiveness than GPLs, and, thus, typically smaller in terms of syntax. For example, SQL's semantics have been formalized by translating queries to relational algebra as early as 1985 [46]. In addition to the dynamic semantics (describing the evaluation of programs), the static semantics (including typing rules) are typically more elaborate in GPLs, for example to take object-oriented programming, subtyping, or higher-kinded types into account [189]. Again, due to their simplicity, DSLs are often not concerned with complex typing rules and have simple type systems.

The standard text books list three basic styles of specifying formal semantics [189, p. 32f][228, p. xix][215, p. vii], listed below. The following short explanations are illustrated with semantic rules for the language constructs `SKIP` (the empty command) and `;` (the sequencing of commands). These two are elements of virtually all imperative programming languages and nicely demonstrate the different viewpoints of the individual semantic styles. For the sake of simplicity, questions of termination, non-determinism etc. are ignored.

- *Operational semantics* defines a *translation function* that maps the terms of the language to equivalent terms. The meaning of a term is then defined as the term that results from following this mapping until no further translations apply (Figure 2.4). Alternatively to this *small-step* semantics, *big-step* semantics are defined with a relation that directly maps a term to its final form (Figure 2.5). To emphasize the

$$\frac{}{(\texttt{SKIP}; C_2, s) \Rightarrow (C_2, s)} \qquad \frac{(C_1, s) \Rightarrow (C_1', s')}{(C_1; C_2, s) \Rightarrow (C_1'; C_2, s')}$$

Figure 2.4.: For small-step semantics, a translation function $\Rightarrow$ maps a pair of a program and an (abstract) machine state to a pair of the remaining program and a changed machine state by taking an atomic computation step. The first rule states that executing a sequence of `SKIP` and some other command C_2 can be continued by just executing C_2. The second rule reduces the first command in a sequence. For example, the first command might be reduced to `SKIP` (possibly in several steps), which makes the first rule applicable.

$$\frac{}{(\texttt{SKIP}, s) \Downarrow s} \qquad \frac{(C_1, s_1) \Downarrow s_2 \qquad (C_2, s_2) \Downarrow s_3}{(C_1; C_2, s_1) \Downarrow s_3}$$

Figure 2.5.: For big-step semantics, a translation function $\Downarrow$ maps a pair of a program and an (abstract) machine state to a final state. The command `SKIP` leaves the state unchanged, which is defined by the first rule. The second rule states that a sequence of two commands C_1 and C_2 is executed by executing C_1, yielding state s_2, and then using s_2 as the initial state for executing C_2, which yields the overall result state s_3.

difference, applying small-step semantics is sometimes called *reduction*, and big-step semantics is called *evaluation* [189, p. 34].

- *Denotational semantics* defines an *interpretation function* that maps language terms to objects of some *semantic domain* (Figure 2.6). For example, SQL queries can be translated to relational algebra expressions [46], which allows proofs about the equivalence of queries.

- *Axiomatic semantics* finally defines laws about the terms of the language. One way to do this is to give Hoare-triples $\{P\}C\{Q\}$, where P is termed the pre-condition, C is the program (fragment), and Q is the post-condition. Such a triple expresses that, in a state in which P holds, the execution of C will yield a state in which Q holds. The semantics of a language can be defined with a collection of such triples (Figure 2.7).

The relation of the definition styles to each other shifted in the last decades. Tennent [228] used the denotational style as the primary approach to defining semantics in 1991, whereas in 2002 Pierce [189] argues that operational semantics surpasses the other styles in flexibility and simplicity. The 2014 textbook by Nipkow and Klein [167] confirms this and focuses on operational semantics as well. In the remainder of this thesis, we will mainly use operational semantics.

$$
\begin{aligned}
[[\texttt{SKIP}]](s) &= s \\
[[C_0; C_1]](s) &= [[C_1]]([[C_0]](s))
\end{aligned}
$$

Figure 2.6.: Denotational semantics of commands are defined by a function $[[\cdot]]$ that maps each command to a function from states to states. As such, the semantics of `SKIP` is the identity function. A sequence of commands is mapped to the composition of the functions defined by the individual commands.

$$
\frac{}{\{P\}\texttt{SKIP}\{P\}} \qquad \frac{\{P\}C_0\{Q\} \quad \{Q\}C_1\{R\}}{\{P\}C_0; C_1\{R\}}
$$

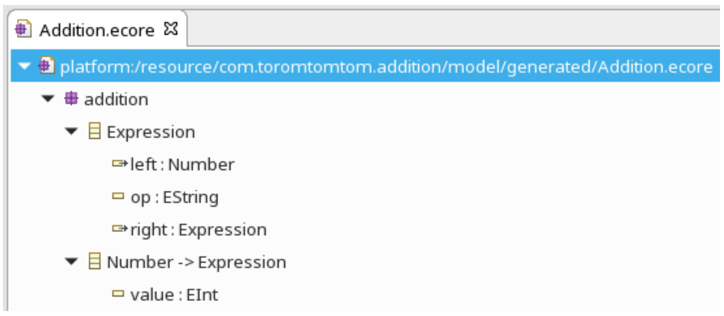
Figure 2.7.: For axiomatic semantics, inference rules about Hoare-triples (pre-condition, program (fragment), post-condition) are defined. The `SKIP` command does not change anything and, thus, pre- and post-condition are equal. Two sequenced commands are glued together by using the post-condition of executing the first command as the pre-condition of executing the second command.

2.5. Designing and implementing domain-specific languages

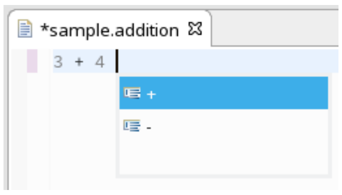
Concrete syntax, abstract syntax, and semantics are the necessary ingredients for GPLs as well as DSLs. However, due to DSLs focusing on a specific application domain, they are often smaller in all three aspects. For that reason, DSLs are sometimes called "little languages" [19]. Consequently, the implementation of a DSL can be expected to be simpler than the implementation of a GPL. One trick of many DSLs is that they only form a thin layer above an existing GPL. We have already studied code generation and embeddings in functional languages, which both heavily rely on a target or host language. The interaction with an existing language is captured in two DSL categories: *external* DSLs are defined independently of any existing language, whereas *internal* DSLs reuse the syntax and semantics of an existing language. We now look into this distinction and how it interacts with the techniques covered so far. There are many previous comparisons of both approaches, for example in the textbook by Fowler [80], some also specific to Scala [15, 7].

2.5.1. External DSLs

The concrete syntax, abstract syntax, and semantics of *external* or *standalone* DSLs can be designed without constraints imposed by a preexisting language. In particular, the language and its implementation are independent. Therefore, the language design can generously adopt and integrate conventions and metaphors of the application domain. This refers to syntax, allowing arbitrary choices of keywords, punctuation, and other structuring elements, as well as semantics, allowing arbitrary evaluation and execution models. However, this design freedom is limited in several ways. First and foremost,


Addition.ecore
platform:/resource/com.toromtomtom.addition/model/generated/Addition.ecore
addition
Expression
left : Number
op : EString
right : Expression
Number -> Expression
value : EInt

(a) The generated metamodel for a simple grammar of arithmetic expressions. Xtext automatically inferred that `Number` is a subclass of `Expression`, encoded by the arrow `->` in the metamodel.



(b) In the generated editor, autocompletion based on the grammar is available.

Figure 2.8.: The metamodel and editor as generated by Xtext.

formal language theory tells us that a language must satisfy certain properties to be parsable by a computer. Ambiguity and context-sensitivity, for example, make employing established efficient parsing algorithms hard or even impossible. A second limitation arises from the tools that support the implementation of parsers. Parser generators and language workbenches only accept certain kinds of grammars and might require unintuitive changes in the language definition just to please the parsing algorithm. Parser combinator libraries, on the other hand, are customizable, but lack the ability to check a given grammar for properties such as ambiguity.

External DSLs often appear in an imperative or object-oriented setting [86]. When object-oriented programmers talk about DSLs, they mostly refer to grammar-based parsing and generating code in their OOP language of choice. This principle is embodied in language workbenches such as Xtext (see Section 2.3.2). The idea of metamodels as used by language workbenches is closely related to the idea of domain models and similar object-oriented design principles [80, p. 160].

As a simple example for implementing an external DSL with Xtext we use a language that consists of additions and subtractions of integers (cf. Section 2.3.2). One way to express the grammar is the following:

```
Expression:
  Number
  | left=Number op=('+' | '-') right=Expression;


Number:
  value=INT;
```

The definition for integer literals `INT` is provided by Xtext.

Using this grammar, Xtext autogenerates an EMF metamodel with the two classes `Expression` and `Number` (Figure 2.8a). The identifiers in the grammar (`left`, `op`, `right`, and `value`) are translated to attribute names in the metamodel. Based on the first alternative of the first rule, `Number` is inferred to be a subclass of `Expression` in the metamodel. Such type relations can be exploited in the language implementation. For example, it is possible to write a polymorphic function for evaluating any `Expression` object, including `Number` objects. Xtext also generates Java classes and interfaces implementing the metamodel.

The metamodel classes can be instantiated by parsing a concrete arithmetic expression. Xtext generates an ANTLR 3 parser and an Eclipse plug-in with an editor based on the given grammar. With Xtext's default settings, the editor already provides some language-specific features, for example autocompletion (Figure 2.8b).

Depending on the language, Xtext can provide a sizable part of the language implementation without any additional work by the language implementer. The language-specific metamodel facilitates implementing features beyond the defaults. Typical example are type checking, scoping rules, and interpretation or compilation. Xtext provides extension points for these features to integrate them seamlessly into the generated Eclipse plug-in. In particular, the generation of Java code is well integrated, as the generated code can be directly compiled in Eclipse. Therefore, mograms in an external DSL in Xtext can be written and executed like in a standard programming language. This makes Xtext (and other language workbenches) a valuable tool for implementing external DSLs.

2.5.2. Internal DSLs

Internal DSLs or *embedded* DSLs are implemented as a part of a host GPL. Whereas the design of external DSLs is based on formal language theory, grammars, and parsing techniques, the idea of internal DSLs can be considered as designing particularly expressive APIs. Thus, the host language's functionality as well as its libraries, tools, and editors can be reused when working with the DSL. Internal DSLs "are just expressions in the host language, written in a form that makes them read well as a language" [80, p. 85]. Consequently, their syntax and semantics must be compatible with the host language, which constrains the language design freedom. How much the language design is constrained depends heavily on the concrete host language. Some GPLs are known to be more suited for the definition of internal DSLs than others, and the ability to define DSLs can be a central feature of a GPL. For example, the Lisp family of languages is known to facilitate the development of embedded languages due to its integration of code and data [80, p. 488]. Similarly, Ruby has many features that benefit the implementation of internal DSLs [55]. Scala is advertised as a language that supports embedded DSLs [171, p.xxxix]. By "bending" existing GPLs this way, internal DSLs can be easily implemented. Another example, applicable to any object-oriented language, is the idea of fluent interfaces from Section 2.2.4.

Internal DSLs often appear in functional programming languages [86]. In Section 2.3.3 we distinguished shallow and deep embeddings; particularly the deep embedding relies heavily on algebraic data types as typically offered in FP languages. Higher-order functions are another feature of FP languages that is very useful in designing shallowly embedded DSLs.

We again use the example of addition and subtraction expressions to illustrate the implementation of internal DSLs. In contrast to the discussion in Section 2.3.3, we now focus on implementing the concrete syntax of an internal DSL. We aim to replace the functions `lit` (for integer literals) and `add` (for addition) with syntactically lighter variants. As the host language we use Scala and demonstrate how features of the host language interact with the language design.

A first improvement is to replace `lit` with `Number`, and then `add` with the infix operator `+`, which is enabled by two features of Scala. First, symbols can be used as method names. Second, methods with one argument can be invoked without a dot `.` and parenthesis `()`. For example, the following two expressions are equivalent:

```
Number(3).+(Number(4)).+(Number(5))
Number(3) + Number(4) + Number(5)
```

Thus, to use the infix operator `+` in the DSL, it suffices to implement in the class `Number` (or a superclass) a method named `+` that takes another `Number` (or a superclass) as its argument.

As a next step, we would like to eliminate the `Number` constructor calls. Ideally, we would just write `3 + 4 + 5`, and be able to construct an abstract syntax tree with each integer wrapped in a `Number` object. To enable the compiler to do thus wrapping automatically, we can define an implicit conversion:

```
implicit def intToNumber(i: Int): Number = Number(i)
```

Then, when the compiler encounters an `Int` value in a place where it requires a `Number`, it uses the implicit conversion to create a `Number` object. However, the compiler can perfectly evaluate `3 + 4 + 5` and, therefore, will not look for implicit conversions here. To force the usage of our `Number` type, we need some syntactical overhead. For example, the following expressions would work:

```
Number(3) + 4 + 5
(3: Number) + 4 + 5
```

This exemplifies the limits of internal DSLs. With the syntax and semantics rules of the host language taking preference over definitions in the language, the design flexibility of external DSLs cannot be achieved. Different features of the host language, for example infix method calls or implicit conversions, can alleviate that, but never close the gap to external DSLs completely. The advantage of internal DSLs is that the implementation does not need to cross language borders and is, therefore, much simpler. In particular, the host language's type system, libraries, and tools can be reused. But then, a DSL might require domain-specific error messages, which might conflict with reusing the host language compiler [209].

2.5.3. Expressive power

The limits in the design of the language syntax are set by formal language theory for external DSLs and by the host language for internal DSLs. However, in both cases there is no intrinsic limit for the language semantics. Although DSLs are defined as languages with inferior formal expressiveness, it is perfectly possible to reimplement all features of a typical GPL when designing a DSL. In fact, one important insight when designing a DSL is to make it as expressive as needed, *but not more*. The usefulness of DSLs is a direct consequence of their reduced expressiveness, as this allows processing DSL mograms in a more straightforward way than GPL programs. Informally speaking, “constraints liberate, liberties constrain” [23]. As a consequence, it can be deduced that a more powerful language allows for less reasoning about mograms in the language.

This quite abstract concept can be illustrated with the relation of defining and calling a function [23]. Consider the following two function signatures:

```
def foo(a: Int): Int

def bar[A](a: A): A
```

The function `foo` is very specific in its type. As a result, calling it is constrained to specific input values; its implementation, on the other hand, is unconstrained and could return essentially every possible `Int` value. In contrast, the function `bar` can be called without constraints. But for exactly that reason, it can not make any assumptions about its input value—it is constrained to one implementation: returning its argument[4]. Freedom for the caller limits the freedom for the callee and vice versa. In the same sense, freedom for the user of a DSL limits the freedom of the DSL implementation.

Another example for trading freedom is referential transparency. Calls to a pure function, that is a deterministic function that does not cause any side effects, are referentially transparent. In a language that only allows pure functions, function call results can be freely memoized and reused. For example, the following snippet can be optimized by a compiler only if the function `foo` is a pure function. Depending on the cost of evaluating `foo(5)`, such optimizations can increase the performance of programs tremendously.

```
// original version
bar(foo(5), foo(5), foo(5))

// optimized version
val foo5 = foo(5)
bar(foo5, foo5, foo5)
```

This exemplifies how not allowing the *language user* to do side effects gives the *language implementer* the chance to transform and optimize the program. In general, constraining the expressive power of the language means more potential for analysis and optimization in the language implementation.

2.6. Summary

In this section we have investigated the three main components of software languages: concrete syntax, abstract syntax, and semantics. Many considerations regarding these aspects apply to GPLs as well as to DSLs. However, DSLs are typically simpler than GPLs, which means that less sophisticated methods are required to implement them, and

[4]Technically, it could also run forever or crash the program and, thus, never return a value at all.

it is simpler to statically analyze a DSL than a GPL. For example, the evaluation order in a declarative DSL can be chosen optimally, whereas statements in an imperative GPL have a fixed order.

Two main approaches to implementing DSLs can be distinguished: external DSLs, which define their own syntax and semantics, and internal DSLs, which reuse syntax and semantics of their host language.

To implement external DSLs, language workbenches are one popular way. For example, the language workbench Xtext takes as input an augmented grammar and generates a parser for handling the concrete syntax, a metamodel to represent the abstract syntax, and extension points to implement semantics (e.g., a type checker, a code generator, or an interpreter). This way, the DSL and the implementation language can be completely separated.

In contrast, internal DSLs are implemented without leaving the host language. Thus, the concrete syntax is represented with host language constructs. In a deeply embedded internal DSL, the abstract syntax is expressed with host language data structures, and the semantics as functions from the abstract syntax to some result. A shallow embedding, on the other hand, directly maps the concrete syntax to some result, omitting an explicit abstract syntax.

The utility of using a DSL is largely independent of whether it is internal or external. Although both variants have different implications regarding the design of the concrete syntax, they both allow encoding domain-specific concepts in the abstract syntax and the semantics. This makes DSLs useful in modeling and simulation, in particular for specifying simulation models based on stochastic processes. Before taking a closer look at such DSLs, we give a short introduction to the underlying math of a specific class of stochastic processes in the next chapter.

3. Modeling population processes

In the previous chapter, we discussed DSLs from a software perspective. However, as established in the introduction, DSLs used for simulation modeling can also be considered as languages that describe mathematical objects. In the following, we look into CTMCs, one class of such objects, and how it can be employed to model and simulate various processes. In particular, we see that complex stochastic processes with infinite state spaces can be specified succinctly, a fact we rely on in the next chapter when we study DSLs for simulation modeling.

3.1. Continuous-time Markov chains

Continuous-time Markov chains (CTMCs) are stochastic processes that are very well suited for simulation modeling for several reasons. First, they can express many of the problems that simulation addresses, for example biochemical reaction networks. "[T]he scientific, engineering, and even financial and humanities literature, abounds with examples of random processes which have been modelled, with varying degrees of success, by Markov chains" [6, p. vii]. Second, they can be efficiently executed using so-called stochastic simulation algorithms (SSAs). Lastly, they can be succinctly specified using different kinds of formalisms [103]. The following definitions are taken from Anderson [6] and Whitt [248]. An overview of alternatives to CTMCs for defining stochastic processes is given in De Nicola et al. [58].

3.1.1. Intuition

Before giving the formal definition of CTMCs, we introduce the intuitive meaning behind it based on an example. We model with a CTMC how some machine, for example a printer, breaks down and gets repaired [248, p. 10f.]. Figure 3.1 shows a *rate diagram* for the example model. The machine has three states: it is working, broken, or getting repaired. A working maching breaks with a rate of 0.05. When broken, the machine gets repaired with a rate of 1.0. The repair can be successful with a rate of 0.9, or fail with a rate of 0.1. The rates are the rates of exponential distributions, and can be interpreted as follows. A state transition with the rate λ occurs λ time per time unit *on average*, or, conversely, it takes $1/\lambda$ time units until the state transition occurs *on average*.

This way, a CTMC describes how a system changes its state through time, where the duration of staying in each state follows some distribution, such as the exponential distributions in the example above. The defining property of CTMCs is the *Markov property*. It states that the future development of the system only depends on its current state, not on its past states. This benefits reasoning about CTMCs, and simulation

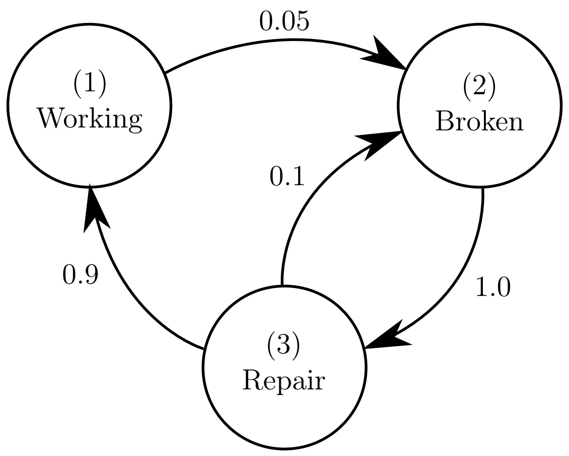


Figure 3.1.: The rate diagram (or intensity graph) of a simple CTMC is a graph, in which the states are the nodes and the possible state transitions are directed edges, labeled with the transition rates

algorithms are able to exploit the Markov property as well. If further the development of the system does not depend on the time already spent in the current state, the CTMC is called *homogeneous*. When using exponential distributions, CTMCs are always homogeneous; however, there exist generalizations of the exponential distribution that can be used in CTMCs that are inhomogeneous. We will not consider inhomogeneous CTMCs any further in this thesis.

Given a CTMC, the behavior of the modeled system can be investigated in several ways. It is possible to analytically determine what proportion of time is spent in each state, assuming that the CTMC runs infinitely long. Alternatively, simulation algorithms sample trajectories through the state space by, given a state, computing the time and destination of the next state transition. In our example, both methods could answer questions such as "what proportion of the total time is the printer working?" Before going into more detail, however, we give some formal definitions for CTMCs and related terms.

3.1.2. Formal definition

Consider a stochastic process $\{X(t), t \in [0, \infty)\}$, where t is the *time* and the $X(t)$ are *states* from a set E (called the *state space*). This process is a CTMC, if for any $0 \leq t_1 \leq t_2 \leq \cdots \leq t_n \leq t_{n+1}$ and corresponding states $i_1, i_2, \ldots, i_n, j \in E$ the *Markov property* holds:

$$P(X(t_{n+1}) = j | X(t_1) = i_1, X(t_2) = i_2, \ldots, X(t_n) = i_n) = P(X(t_{n+1}) = j | X(t_n) = i_n)$$

As described above, this equality states that the future development of the process depends only on its current state, not on its past states. Homogeneity can be defined as follows: A CTMC is *homogeneous*, if for any $0 \leq s \leq t$ and $i, j \in E$

$$P(X(t) = j|X(s) = i) = P(X(t - s) = j|X(0) = i).$$

This states that the probability for any state transition is independent of how long the system already is in the current state. The right side of that equation can be transformed to the *transition function* of the CTMC:

$$P_{i,j}(t) = P(X(t) = j|X(0) = i) \text{ for } i, j \in E, t \leq 0$$

The transition function tells us how probable it is to find the system in state j when, t time units earlier, it was in state i. This includes transitioning through intermediate states before reaching j. To fully define a CTMC, we additionally need a distribution about the probabilities of all states $j \in E$ at $t = 0$, denoted as $P(X(0) = j)$. Then, the probability to be in any state $j \in E$ at time t is

$$P(X(t) = j) = \sum_{i \in E} P(X(0) = i)P_{i,j}(t).$$

This expresses that the probability of being in state j after time t is the sum of all probabilities of being in state i at time $t = 0$ and transitioning from i to j in t time units, maybe passing through intermediate states.

Instead of handling the transition function directly, often its derivative Q is used. One way to define it is via the infinitesimal differential:

$$Q_{i,j} = \lim_{h \to 0} \frac{P_{i,j}(h) - P_{i,j}(0)}{h}$$

where

$$\begin{aligned} &P_{i,i}(0) = 1 \\ &P_{i,j}(0) = 0 \text{ for } i \neq j, \end{aligned}$$

meaning that no state transitions occur without time passing. By interpreting Q and P as matrices, the same derivation can be expressed as $Q = P'(0)$. Q is called the *q-matrix*. The example from Figure 3.1 is captured in the following q-matrix, where each diagonal element is the negative of the sum of the non-diagonal elements of that line. The non-diagonal elements are the transition rates.

$$Q = \begin{bmatrix} -0.05 & 0.05 & 0 \\ 0 & -1.0 & 1.0 \\ 0.9 & 0.1 & -1.0 \end{bmatrix}$$

3.1.3. Balance analysis

After defining CTMCs formally, the task of analyzing a given CTMCs can be formalized as well. For example, we might want to know what distribution of states emerges in the (infinitely) long run. This distribution can, under certain conditions, be determined by solving a system of linear equations [6, p. 160f.]. We represent such a distribution by listing the probabilities for all individual states $\alpha_i, i \in E$ in a line vector $(\alpha_1, \alpha_2, \dots)$. The $\alpha_i, i \in E$ must sum up to 1:

$$\sum_{i \in E} \alpha_i = 1$$

A steady-state or stationary distribution can then be formally defined as a distribution that does not change when the CTMC continues to run:

$$\sum_{i \in E} \alpha_i P_{i,j}(t) = \alpha_j \text{ for all } j \in E, t \geq 0$$

We can also describe the steady-state distribution via the q-matrix (the derivative of the transition function) by stating that the change is 0:

$$\alpha Q = 0$$

Applying this equation to the example yields the following system of linear equations:

$$\begin{aligned}
-0.05\alpha_1 \qquad\quad +0.9\alpha_3 &= 0\\
0.05\alpha_1 - 1.0\alpha_2 + 0.1\alpha_3 &= 0\\
+1.0\alpha_2 - 1.0\alpha_3 &= 0\\
\alpha_1 + \quad \alpha_2 + \quad \alpha_3 &= 1
\end{aligned}$$

With the unique solution $\alpha_1 = 0.9$, $\alpha_2 = \alpha_3 = 0.05$. We can conclude that the machine in our example model is working 90% of the time.

3.1.4. Stochastic simulation algorithms

The analytical approach to finding a steady-state distribution only works "for Markov chains with a very simple structure" [103, p. 19]. For complex CTMCs, the steady-state distribution can be obtained analytically only by approximation (see, for example, Dayar et al. [57]).

Alternatively, the CTMC can be explored with stochastic simulation algorithms (SSAs). Examples for SSAs are the so called Doob-Gillespie algorithms [62, 88, 89]. These algorithms sample a trajectory from the CTMC defined by the model by repeating a simple step. Given the current simulation time t and the current model state i the algorithms determine the time of the next state change t' and the next state i' following the CTMC's distribution. The initial state i_0 of the model at time $t = 0$ is given by an initialization. The variations of the stochastic simulation algorithm differ in the implementation of this step.

One of the basic SSA variants originally introduced by Gillespie is the First Reaction Method. In the First Reaction Method the basic SSA-step described above consists of three parts:

1. Sample a waiting time Δt_j for each state $j \neq i$ that is not the current state i. As waiting times follow an exponential distribution with parameter $q_{i,j}$, this can be efficiently done using the inversion method:

$$\Delta t_j = \frac{1}{q_{i,j}} \ln \frac{1}{r}, \qquad r \sim \text{uniform}(0,1)$$

2. Find the state i' with minimal Δt_j.

$$i' = \operatorname*{argmin}_j \Delta t_j$$

3. Set t to $t + \Delta t_{i'}$ and set the current state to i'.

The *First Reaction Method* directly implements the idea of the so-called *stochastic race*. All possible transitions compete against each other and the one with the shortest waiting time drawn wins. Transitions with higher transition rates $q_{i,j}$ are chosen with higher probability.

For example, if in our example the state at time t is “Repair”, the successor states are “Working” (with a rate of 0.9) and “Broken” (with a rate of 0.1). The time of transitioning to the next state is the minimum of $\Delta t_{\text{Working}}$ and Δt_{Broken}. We draw numbers from exponential distributions for both: $\Delta t_{\text{Working}} = 1.53$ and $\Delta t_{\text{Broken}} = 3.06$. Thus, the next state is “Working” and the transition occurs at $t + 1.53$ time units.

Alternatively to the First Reaction Method, Gillespie proposed the *Direct Method*. The Direct Method exploits that the minimum of the Δt_j is itself exponentially distributed with the rate $\sum_{i \neq j} \Delta t_j$. This way, the overall waiting time $\Delta t_{i'}$ can be drawn directly. The successor state is then chosen according to the proportion of the transition rate to the sum of the rates of all possible transitions:

$$P(\mu = j) = \frac{q_{i,j}}{\sum_{k \neq i} q_{i,k}}$$

For example, if again the state at time t is “Repair”, the sum for all rates for transitions leaving the state is 1.0. Thus, $\Delta t_{i'} \sim \text{Exp}(1.0)$. We draw the value $\Delta t_{i'} = 1.53$, and then select the successor state according to their rate’s proportion of the sum of all rates: $P(\text{Working}) = {}^{0.9}/_{0.9+0.1}$ and $P(\text{Broken}) = {}^{0.1}/_{0.9+0.1}$. When selecting “Working”, we obtain the same successor state and transition time as with the First Reaction Method.

3.2. Markov Population Models

Markov Population Models are a particularly popular variant of CTMC models for simulation [103]. In the simplest case, they describe the development of the size of a

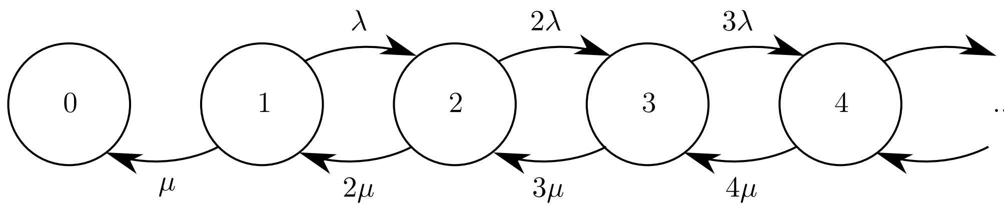


Figure 3.2.: A snippet of the (infinite) rate diagram of a simple birth-death-process. The state transition from 0 to 1 has the rate 0 and is omitted in the graph.

single population, which only changes in steps of 1. That is, each state $n = 0, 1, 2, \dots$ has a transition to the state $n + 1$ (a birth) with the rate λ_n and a transition to the state $n - 1$ (a death) with the rate μ_n. μ_0 is commonly set to 0 (a population size can not become negative).

One way to model a population of entities, where each can proliferate or die, is to factor the population size into the rate [6, p. 62f.]. Thus, in the birth-death-process as described above, $\lambda_n = \lambda n$ and $\mu_n = \mu n$ for some constants λ and μ. The reasoning behind this form of transition rate is based on chemistry and, therefore, explained later in Section 3.3. We call constants like λ and μ *rate constants* and the product of the rate constant with the population size the *propensity* of the state transition. Consequently, for this simple model the propensities are precisely the positive elements of the q-matrix:

$$q_{i,j} = \begin{cases} \lambda i & \text{if } j = i + 1 \text{ and } i \geq 1 \\ \mu i & \text{if } j = i - 1 \text{ and } i \geq 1 \\ -(\lambda + \mu) i & \text{if } j = i \\ 0 & \text{otherwise} \end{cases}$$

Figure 3.2 shows the rate diagram for this definition. Note that, although the CTMC's state space is infinite, the description is finite, and even quite succinct.

More generally, population models can include n distinct species. The state space is then $\mathbb{Z}_+^n$, containing tuples with n non-negative integers, where the i-th integer is the population size of the i-th species. As the individual species can interact in various ways, multi-dimensional population processes are a powerful modeling tool. For example, consider a two-dimensional population model with states (m, n), in which m is the size of a predator species and n is the size of a prey species [6, p. 324ff.]. Then we can define (omitting the diagonal and the 0 matrix elements):

$$q_{(m_1,n_1),(m_2,n_2)} = \begin{cases} \lambda n_1 & \text{if } m_1 = m_2 \text{ and } n_1 + 1 = n_2 \text{ and } n_1 \geq 1 \\ \beta m_1 & \text{if } m_1 - 1 = m_2 \text{ and } n_1 = n_2 \text{ and } m_1 \geq 1 \\ \alpha m_1 n_1 & \text{if } m_1 + 1 = m_2 \text{ and } n_1 - 1 = n_2 \text{ and } n_1 \geq 1 \end{cases}$$

This describes three types of state change: the birth of a prey entity, the death of a predator entity, and a predator entity feeding on a prey entity and proliferating in the process. For certain choices of λ, β, and α, this system exhibits periodic behavior. However, as soon as the predator (or prey) species dies out, the system degrades to a pure birth (or death) process (see Figures 3.4a and 3.4b). Figure 3.3 shows the rate diagram for this CTMC.

3.2.1. Modeling formalisms

The matrix notation that we used to describe population models and CTMCs in general is only one way of specification. In the same way we described the matrix elements with a few expressions, tailored modeling formalisms can be used to specify a population model in a more abstract way. Henzinger, Jobstmann, and Wolf [103] give an overview that we summarize here. The decisive quality of these formalisms, including the above description of the matrix, is that they define *transition classes* of state transitions instead of individual state transitions. For example, in the prey-predator-model, the matrix description does not describe a transition for one particular state. Instead, by ranging over all matrix elements, it describes all state transitions that occur with three expressions.

These modeling formalisms are essentially software languages according to our definition in the previous chapter. Although Petri Nets are a graphical language, syntax and semantics can be defined as we discussed for software languages. In particular, the state space of a stochastic Petri Net with exponentially distributed firing times is a CTMC, and populations can be represented by a specific place, with the number of tokens encoding the population's size. Similarly, stochastic process algebras or stochastic guarded commands are languages that can be used to specify population models. As textual languages with origins in computer science, formal definitions of syntax and semantics are a natural aspect of these approaches. A more domain-specific approach are stoichiometric equations, which adopt notational conventions from biochemical reactions (see Section 3.3). First, we consider two more modes of evaluating CTMCs that are tailored to population models.

3.2.2. Next Reaction Method

As an extension to the First Reaction Method, Gibson and Bruck proposed the Next Reaction Method [87]. It exploits the fact that in many population models state transitions are caused by parallel, independent processes. One such process corresponds to one transition class. For example, in Figure 3.3, the state transitions for prey proliferating (an upward arrow) and predators dying (an arrow to the left) can be considered independent. If in a given state both transitions are possible and one of them actually occurs, the other is still possible with the same rate as before.

For example, in the state $(2, 2)$ a prey proliferates with the rate 2λ and a predator dies with rate 2β. If now the transition caused by proliferation of prey occurs, the successor state is $(2, 3)$. In this new state, the rate for a predator dying is still 2β. If the transition caused by the death of a predator fires first, the rate for prey proliferation remains unchanged.

The Next Reaction Method makes use of this observation by not re-evaluating all rate expressions after a state transition. Instead, the firing times that have been drawn for state transitions are kept in an event queue and only updated if the value of the corresponding rate expression changed. To determine which rate expressions have changed, a dependency

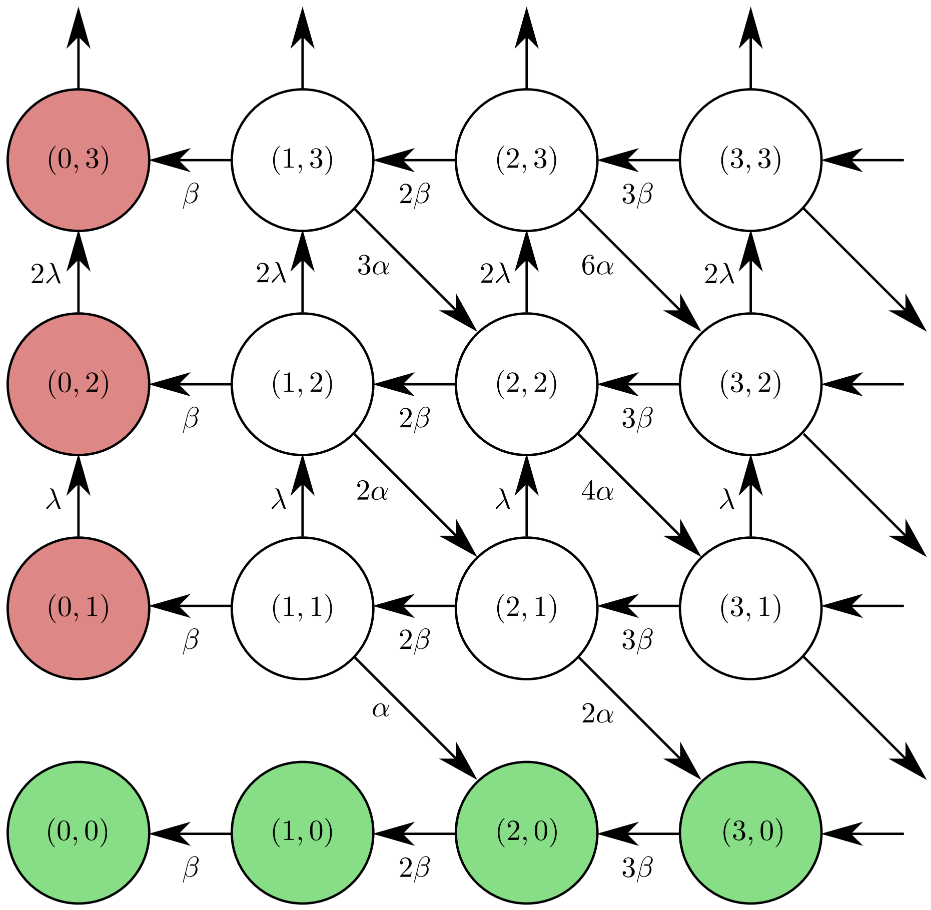


Figure 3.3.: A snippet of the (infinite) rate diagram of a simple two-dimensional predator-prey-model. Proliferating prey is represented by an upwards state transition, the death of a predator by a transition to the left. A predator feeding on a prey is represented by a transition to the lower right. In the states colored green the prey species has died out, and the system degraded to a death process of the predator species. In the states colored red the predator species has died out, and a birth process of the prey species remains.

graph between classes of state transitions is maintained. Identifying these classes relies on the modeling formalism that generated the CTMC. In the dependency graph, a class of state transitions B depends on another class A, if executing a state transition in A makes re-evaluating the rates of state transitions in B necessary. For example, in the predator-prey model in Figure 3.3 the rate of a predator feeding on a prey (an arrow to the lower right) must be re-evaluated if a prey proliferates (an upward arrow) or a predator dies (an arrow to the left).

Depending on the model, the number of dependencies can be much smaller than the number of transition classes, which allows for more efficient simulation in comparison to the Direct or First Reaction Method. The Optimized Direct Method [42] further increases the simulation efficiency. It also adopts the idea of using a dependency graph. However, the Optimized Direct Method only saves the values of the transition rates instead of the firing times, saving the overhead for managing an event queue.

3.2.3. ODE approximation

The SSAs we have studied so far are exact, in that they produce a statistically accurate sample path through the CTMC [91]. However, their performance heavily depends on the population sizes in the model. In particular, larger population sizes usually lead to a higher frequency of reactions, meaning that the simulation progresses more slowly. There are SSA variants that sacrifice exactness to speed up the simulation in several ways.

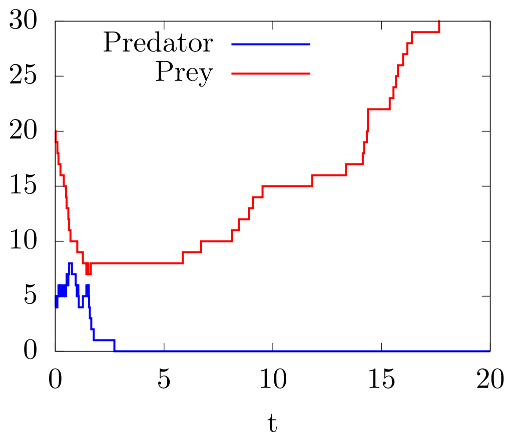
As an alternative, it is possible to express the changes of the population sizes as a system of ordinary differential equations (ODEs) [129]. This can then be evaluated numerically with an ODE solver, leading to a deterministic interpretation of the CTMC. The system of ODEs is called the *reaction-rate equation* and defined as [91]:

$$\frac{dX_i(t)}{dt} = \sum_j v_{i,j}\alpha_j(X(t)),$$

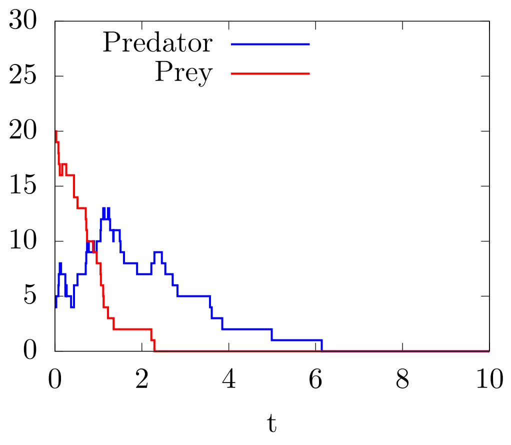
where $X(t)$ is the state vector at time t, $v_{i,j}$ is the i-th element of the change vector for the j-th state transition class, and $\alpha_j(\cdot)$ is the state-dependent rate of the j-th state transition class. The change vector describes the changes caused by the transitions in a transition class as the difference in each population size.

The predator-prey example can be reformulated in this approach as follows:

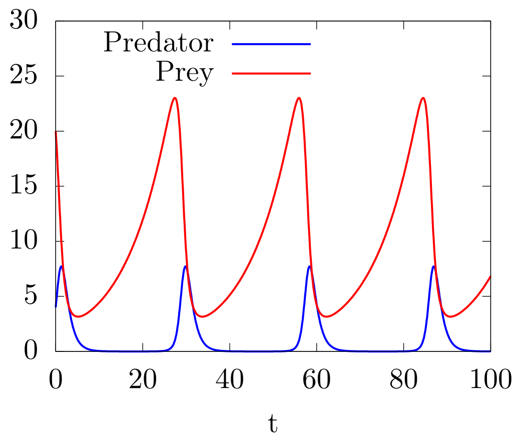
	$X_1 = m$	$X_2 = n$
$\alpha_1((m,n)) = \lambda n$	$v_{1,1} = 0$	$v_{2,1} = 1$
$\alpha_2((m,n)) = \beta m$	$v_{1,2} = -1$	$v_{2,2} = 0$
$\alpha_3((m,n)) = \alpha m n$	$v_{1,3} = 1$	$v_{2,3} = -1$



(a) A stochastic simulation run in which the predator species dies out. Subsequently, the prey population grows exponentially.



(b) A stochastic simulation run in which first the prey species and then the predator species dies out.



(c) A numerical solution of the reaction-rate equation. The predator population and the prey population oscillate.

Figure 3.4.: Three plots of simulation runs for a predator-prey-model with $\alpha = 0.1$, $\beta = 1.0$, and $\lambda = 0.1$. Initially, there are 20 prey and 4 predator entities.

$$\frac{dX_1(t)}{dt} = \frac{dm(t)}{dt} = 0\lambda n - 1\beta m + 1\alpha mn = -\beta m + \alpha mn$$
$$\frac{dX_2(t)}{dt} = \frac{dn(t)}{dt} = 1\lambda n + 0\beta m - 1\alpha mn = \lambda n - \alpha mn$$

A numerical solution of this system of ODEs is plotted in Figure 3.4c. It shows the typical periodic behavior of both populations. More prey leads to more predators, and more predators lead to less prey.

The approximation of a CTMC with a system of ODEs produces different behavior than an exact simulation with an SSA. First, the ODE approximation is deterministic, whereas the SSA is stochastic. Second, the population sizes in the SSA are discrete, whereas the ODE approximation uses continuous variables. This leads to the third difference: in the SSA simulation, one of the populations can die out, which leads to a degraded model behavior[1]. In the ODE approximation, however, this is not possible. In general, the ODE approximation is imprecise if small population sizes occur in the model. When a model contains population sizes that are "not too many orders of magnitude larger than one, discreteness and stochasticity may play important roles" [91, p. 36].

3.3. Biochemical reaction networks as population processes

We already introduced CTMCs and their subclass MPMs. We now study how MPMs can be used for modeling biochemical reaction networks (BRNs) based on stoichiometric equations. But first, we introduce some terminology.

First, we assume that biochemical *reactions* occur in a (chemical) *solution*. A reaction can be described by stoichiometric equations of the general form

$$\text{Reactants} \xrightarrow{\text{rate}} \text{Products},$$

where reactants and products are potentially empty lists of biochemical *species*, and the rate is a real number. Reactions are a natural and established representation of biochemical processes.

There are two different ways to relate solutions and species. A solution can be interpreted as a mapping of each species to a concentration. Alternatively, a solution can be interpreted as a mapping of each species to the discrete number (count) of *entities* present. Whereas integer entity counts lead to the discrete states of MPMs, real-valued concentrations correspond to the ODE approximation as discussed in Section 3.2.3. In the remainder of this thesis, we will be concerned with entities rather than concentrations. As a last term, we call the entirety of entities of the same species in a solution a *population*.

A model's behavior is then defined by a list of reactions. For example, the predator-prey model from the Section 3.2 can be expressed as follows[2]:

[1]It can actually be shown that one of the species will die out with probability 1 if the process runs infinitely long [6, p. 325].

[2]We ignore the fact that these are not *chemical* reactions.

$$\text{Prey} \xrightarrow{\lambda} 2\text{Prey}$$
$$\text{Predator} \xrightarrow{\beta} \emptyset$$
$$\text{Predator} + \text{Prey} \xrightarrow{\alpha} 2\ \text{Predator}$$

As shown in Section 3.2, the rates for the transitions from a specific state also depend on the population sizes in that state. Thus, the rate of a reaction results from multiplying the constant above the reaction arrow with the reactant population sizes. We now give the general reasoning behind these *mass action kinetics* [103][90, p. 356ff.].

Chemical solutions are often assumed to be be "well-stirred", meaning that the chemical entities (molecules) are uniformly distributed. When we model populations of molecules, we assume that the molecules move randomly in some confined volume. The probability for two specific molecules to react in some infinitesimal time interval is then the product of the probability of those molecules colliding and the probability of a reaction, given a collision. A similar argument can be made for reactions with more reactants, and reactions with one reactant do not require collision at all. For a constant volume the probability for a specific reaction to occur within the next infinitesimal time interval is proportional to the number of *distinct combinations of reactants* in the current state.

Thus, for a reaction of the form

$$c_1 X_1 + \ldots + c_n X_n \xrightarrow{r} \ldots,$$

where c_i is the number of required entities of species X_i, and a state

$$(s_1, \ldots, s_n),$$

where s_i is the number of available entities (population size) of species X_i, the rate for a state transition according to this reaction is

$$r \cdot \prod_{i=1}^{n} \binom{s_i}{c_i}$$

If the volume Ω is to be taken into account, the formula changes to

$$r \cdot \prod_{i=1}^{n} \binom{s_i}{c_i} \cdot \Omega^{1-\sum_{i=1}^{n} c_i}$$

Mass action kinetics are the fundamental assumption behind modeling BRNs as MPMs, and also allow succinct formulation of such models. However, there is a subtle ambiguity that occurs when several entities of the same population react.

3.3.1. Calculating the reaction rate

We now consider the calculation of reaction rates according to mass action kinetics in more detail. For simplicity, we focus on second-order reactions, that is reactions with two reactants. A *heteroreaction* is a reaction with reactants from distinct populations, whereas in a *homoreaction* two entities from the same population react [68]. As mentioned in Section 3.3, biochemical reactions can be formulated and evaluated as differential equations (modeling continuous concentrations) and as discrete-event systems (modeling individual molecules). Surprisingly, the translation between both approaches is different for heteroreactions and homoreactions.

Heteroreactions

We first consider the following reaction as a reaction of concentrations.

$$A + B \rightarrow A + B + C.$$

This reaction does not modify the concentration of A and B, whereas the concentration of C increases linearly. The evolution of the concentration $c(t)$ of C over time can be solved. The concentrations $a(t) = a_0$ and $b(t) = b_0$ remain constant. For simplicity we assume an initial concentration $c(0) = 0$. The *reaction rate constant* k describes how fast this reaction occurs.

$$\begin{aligned} \frac{dc(t)}{dt} &= ka(t)b(t) = ka_0b_0 \\ c(t) &= ka_0b_0t \end{aligned}$$

Alternatively, the reaction can be expressed as a discrete-event system in which one A and one B react with a *stochastic rate constant* r. We can then describe the evolution of the mean population size of $C(t)$ over time as a differential equation [90, p. 353]. Again the population sizes $A(t) = A_0$ and $B(t) = B_0$ remain constant, and we assume that $C(0) = 0$. The stochastic rate constant r is multiplied with the number of distinct reactant combinations:

$$\begin{aligned} \frac{dC(t)}{dt} &= rA(t)B(t) = rA_0B_0 \\ C(t) &= rA_0B_0t \end{aligned}$$

As both approaches describe the same process, we want the evolution of $c(t)$ and $C(t)$ to be identical. To express this, we translate between concentration and population size by assuming a constant factor γ, which yields $\gamma s(t) = S(t)$ for any species S. A typical choice for γ would be $\gamma = VN_A$ (the surrounding volume multiplied with the Avogadro constant) [43]. This way, the population size is interpreted as the number of particles per volume.

$$\begin{aligned}\gamma c(t) &= C(t)\\ \gamma k a_0 b_0 t &= r A_0 B_0 t\\ \gamma k \frac{A_0}{\gamma}\frac{B_0}{\gamma} t &= r A_0 B_0 t\\ r &= \frac{k}{\gamma}\end{aligned}$$

Thus, we can now determine a stochastic rate constant r that corresponds to an equivalent reaction rate constant k.

Homoreactions

Now consider the reaction

$$A + A \to A + A + E$$

Expressing this via concentrations yields an equation similar to the one for heteroreactions. Again, the concentration of A is constantly a_0 and the initial concentration of E is $e_0 = 0$

$$\begin{aligned}\frac{de(t)}{dt} &= k a(t) a(t) = k a_0 a_0 = k a_0^2\\ e(t) &= k a_0^2 t\end{aligned}$$

Expressing this reaction as a discrete-event system, however, leads to a different formulation for the number of distinct reactant combinations.

$$\begin{aligned}\frac{dE(t)}{dt} &= r\frac{A(t)(A(t)-1)}{2} = \frac{r}{2}A_0(A_0-1)\\ E(t) &= \frac{r}{2}A_0(A_0-1)t \approx \frac{r}{2}A_0^2 t\end{aligned}$$

Now the requirement that both descriptions produce an identical evolution leads to a different factor between k and r as well.

$$\begin{aligned}\gamma e(t) &= E(t)\\ \gamma k a_0^2 t &= \frac{r}{2}A_0^2 t\\ \gamma k \frac{A_0^2}{\gamma^2} t &= \frac{r}{2}A_0^2 t\\ r &= 2\frac{k}{\gamma}\end{aligned}$$

Now, r is twice as large as for a heteroreaction.

3.3.2. Relating rate interpretations

We have seen four descriptions of a system in time. It is sensible to expect all four formulations to produce the same behavior, given the same initial conditions. This expectation is fulfilled for concentration-based formulations, where the same reaction rate constant k produces the same behavior in hetero- and homoreactions. In the discrete-event paradigm, however, where the number of distinct reaction combinations is factored into the reaction propensity, the stochastic rate constant of a second-order homoreaction needs an additional factor 2. More generally, for a reaction as defined in Section 3.3 the factor is $\prod_{i=1}^{n} c_i!$. The rate is then no longer multiplied with the binomial coefficient $\prod_{i=1}^{n} \binom{s_i}{c_i}$, but with the "falling factorial" [106] or "m-fold falling product" [176]: $\prod_{i=1}^{n} [s_i(s_i - 1) \dots (s_i - c_i + 1)]$.

Concretely, when writing two reactions with the same reaction rate constant k

$$A + B \xrightarrow{k} A + B + C$$

$$A + A \xrightarrow{k} A + A + E$$

both reactions will be equally fast in the concentration interpretation. Thus, C and E are produced equally fast. In the discrete-event interpretation, however, the first reaction is twice as fast as the second one. Thus, C is produced twice as fast as E.

In a concrete modeling formalism as the ones mentioned in Section 3.2.1, there are different ways to express and evaluate the reaction propensity.

- The language might just use the formulas as shown above. The modeler is required to manually multiply the rate of each homoreaction with 2 if she wants it to have the same propensity as a corresponding heteroreaction.
- The language might not automatically assume mass action kinetics, but instead require the modeler to explicitly specify the rate in dependence on the population sizes. As we will see in the next chapter, this is the approach taken by ML-Rules.
- Alternatively, the implementation of a modeling language could automatically detect homoreactions and modify their rate accordingly. Then the rules for what transformations are applied and when they are applied must be made clear to the modeler. Such rules are often associated with the term "symmetry"[3] and can be established formally, for example with graph patterns. This approach is taken in the modeling language BioNetGen [221].
- Lastly, a simulation algorithm can factor in the 2 indirectly by creating two reactions for each reactant pair. This, for example, happens naturally in process algebra approaches [43]. Again, this means that the rules for how reactions are instantiated must be kept in mind when writing rate expressions.

[3]The Kappa Language User Manual contains a detailed discussion of symmetry and its implications for rate calculation [34]. However, the language implementation does currently not contain an automated handling of symmetries.

These options differ in how explicit they make the calculation of the final reaction propensity. More implicit approaches lead to simpler rate expression, but also require the modeler to know how the rate expression is interpreted. Rule-based modeling languages as introduced in the next chapter make this even more complex, as there a single reaction rule might encode several reactions, which can simultaneously include homo- and heteroreactions [207]. Without going into much detail yet, here is a simple example. Consider a rule

$$C(v_1) + C(v_2) \xrightarrow{k_{xy}} C(v_1 + v_2)$$

that merges two C entities and adds up their attributes values (modeling the entities' volume, for example). The placeholder variables v_1 and v_2 can take equal or distinct values. If they take equal values, both reactants are equal as well and we have a homoreaction. If they take distinct values instead, the reactants are distinct and the reaction is a heteroreaction. Depending on which of the approaches above has been chosen, these cases might need to be handled differently by the modeler or the formalism. We shortly revisit this issue in Section 6.1.1.

3.4. Summary

This chapter gives a short introduction to CTMCs, which are a valuable tool for simulation modeling of biochemical reaction networks. One reason is that population processes can be mapped very well to CTMCs. Another reason is the simplicity of CTMCs. A CTMC can often be described with a small set of state transition rules.

Despite (or maybe because of) the simplicity of CTMCs, care must be taken when translating from a modeling language to a CTMC. There is no consensus on a "right" way of determining the rate for a state transition from the population sizes in the source state, which might include instantiating several equivalent reactions. Established modeling languages have taken different approaches and make different assumptions. This holds the danger of ambiguities and misinterpretations about how a language implementation executes a given model, which in turn can lead to wrong results from simulation studies.

One way to avoid ambiguities is to equip a language with a formal semantics. Then it is not important to be familiar with the language implementation, and models can be reasoned about on a more abstract level. Before looking into formal definitions, however, we give an informal overview about modeling language concepts in the next chapter.

4. ML-Rules and Biochemical Reaction Networks

In the previous chapter we have seen how biochemical reaction networks (BRNs) can be modeled as continuous-time Markov chains (CTMCs). Several DSLs for modeling have adopted this idea. This chapter revolves around ML-Rules, a modeling language for dynamically nested BRNs first proposed in 2011 [149]. We study ML-Rules and other modeling languages in that domain as well as their relation.

4.1. Modeling biochemical reaction networks with ML-Rules

In the following, we introduce a model of the Wnt/β-catenin signaling pathway as presented by Lee et al. [136] and transformed to a stochastic model by Mazemondet et al. [150]. The Wnt/β-catenin signaling pathway plays an important role for the proliferation and differentiation of human neural progenitor cells. It includes a number of cascading processes that start outside of the cell and lead to effects in the cell core. In contrast to later extensions [95], the model we consider here distinguishes only three locations: the space outside the cell, the cell itself (the *cytosol*), and the cell core (the *nucleus*). We will use this simple example model as a recurring example in the next chapters.

The model consists of 5 species and 12 reactions. An initial amount of extracellular Wnt leads to the dephosphorylation of AxinP in the cell, yielding Axin. As AxinP degrades β-catenin, the extracellular Wnt induces an increase of β-catenin in the cell. Subsequently, β-catenin shuttles into the cell's nucleus and triggers the production of Axin in the cell. The Axin is phosphorylated and, thus, β-catenin is again increasingly degraded. As the extracellular Wnt degrades over time, this negative feedback loop negates the effect of the Wnt, and the model goes into a steady state. These species and reactions can be visualized as a network (Figure 4.1).

Based on this example model, we will now introduce ML-Rules and its features. As this means that we move from the abstract, formal space to a concrete technical implementation, we use a mono-spaced font to denote ML-Rules snippets.

4.1.1. Reaction-based modeling

The first approach to modeling a biochemical system as a CTMC is just enumerating all chemical species in the system as well as the reactions between them (see Section 3.2). This level of abstraction is the basis of the systems biology markup language (SBML), which is the de-facto standard exchange format in the systems biology community [107]. For example, SBML is one of the languages that is used to store and curate biological

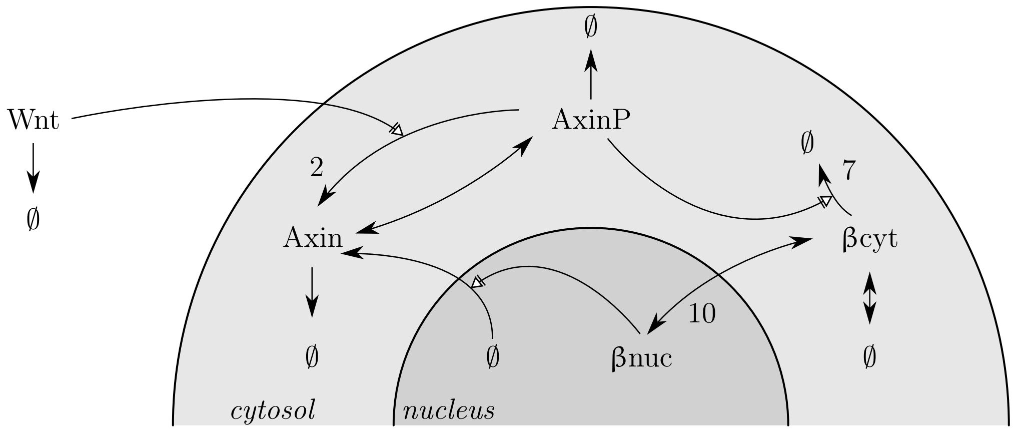


Figure 4.1.: A visualization of the species and reactions in the Wnt/β-catenin model. Plain arrows denote reactions, and arrows with a bar behind a white arrow head denote reactants that are not consumed, but still required for the reaction to occur ("necessary stimulation" [135]). Double-ended arrows denote reversible reactions. The symbol ∅ is used for reactions that either have no reactants or products. Of the 12 reactions, we marked the reactions 2, 7, and 10. The enumeration of the reactions is taken from the publication by Mazemondet et al. [150].

models in the BioModels database [147]. As SBML is primarily designed to be machine-readable rather that human-readable, it does not provide an interface for creating and editing models and is not a modeling language per se. However, SBML model definitions can be imported and exported by various tools that allow for performing modeling tasks (including graphical tools [135]).

The following ML-Rules snippet specifies some species and the reaction 2 from the model above:

```
Wnt(); Axin(); AxinP(); //declare species
Wnt:w + AxinP:a -> Wnt + Axin @ k2*#w*#a; // reaction no. 2 -> rate constant k2
```

This snippet shows a first difference between the more abstract paradigm and the implementation in ML-Rules. ML-Rules is not fixed to mass action kinetics and requires explicitly factoring the population sizes into the rate. Variables (`w`, `a`) can be bound to the reactants, and `#w` and `#a` are then expressions that evaluate to the population sizes. Thus, this reaction has a higher propensity if more `Wnt` is available, although no `Wnt` is consumed. In particular, this reaction does not occur at all if `Wnt` is absent, because in that case the rate is 0. This models the necessary stimulation as described in Figure 4.1.

4.1.2. Attributed entities and rule-based modeling

To be able to design models with a higher level of detail, the concept of attributed entities has been introduced (also called multi-state entities [220]). Here, entities have a number of attributes with values from specific value domains, giving each entity a local state. For example, in the Wnt model the species Axin and AxinP are two states of the same protein with a phosphorylation site. It can also be modeled as a species with an attribute that can take on values representing "phosphorylated" and "not phosphorylated".

```
Wnt(); Axin(bool); //declare species
```

Now each Axin entity will have an attribute which has one of the values `true` or `false`. Thus, `Axin` becomes `Axin(false)` and `AxinP` becomes `Axin(true)`.

In the example model it is easy to make this translation in both directions. In general however, this poses a challenge for reaction-based modeling languages, as each attributed entity can now have multiple realizations. If a specific attribute is irrelevant for a reaction, the reaction would still need to be defined for each value of this attribute. If species have more attributes and several attributed species interact, this leads to a combinatorial explosion.

This shortcoming has led to the development of rule-based modeling languages, in which reaction rules rather than reactions are specified [72]. The left hand side of a reaction rule can contain variables, to which concrete values get assigned during pattern matching against the actual solution. These variables can then be used to define the rate and the products. Consequently, each reaction rule can lead to instantiating a number of reactions, avoiding the combinatorial explosion. On the language design side, the concept of reaction rules also provides means to manage the complexity of the modeled system. By adopting a "don't-care-don't-write" attitude, reaction rules allowing omitting the context of the reaction as much as possible and focusing on the reaction described by the rule. The Wnt-stimulated Axin dephosphorylation in reaction 2 can now be expressed with concrete attribute values.

```
Wnt:w + Axin(true):a -> Wnt + Axin(false) @ k2*#w*#a;
// Wnt:w + Axin(true, x):a -> Wnt + Axin(false, x) @ k2*#w*#a;
```

Below the actual rule, the snippet shows how the rule could look like if `Axin` had a second attribute that remains unchanged by the reaction. Here, `true` and `false` are proper values, whereas `x` is a variable. This second artificial rule demonstrates that the number of necessary rules is independent of the possible values of the second attribute. Consequently, attributes could have infinite domains, which is not possible in the pure reaction-based paradigm.

Attributes in ML-Rules do not have canonical names and instead are identified by their position in the list of attributes. Thus, in a reaction of two entities of the same species, their attributes can be assigned different names. In the following (imaginary) rule, `x1` and `x2` refer to the same attribute of the `Axin` entities `a1` and `a2`, respectively:

```
Axin(x1):a1 + Axin(x2):a1 -> ...;
```

Other rule-based languages assign names to attributes, which allows omitting unchanged attributes instead of repeating them on both sides of the reaction rule. In terms of programming language concepts, ML-Rules' entities with positional attributes can be compared to tuples, whereas entities named with named attributes correspond to records [189, pp. 128ff.].

Rule-based modeling approaches can also be distinguished regarding the effect of a reaction. Whereas ML-Rules supports reactions that create or remove entities from a solution, it also allows to model that entities bind to each other. Thus, some of the reactant's attributes are modified to denote that they are connected as a result of the reaction. For example, in the following snippet two `A` entities are linked by assigning a new, unused, and unique shared value (generated by `nu()`[1]) to one of their attributes.

```
A(link,link);
A(l,free):a1 + A(free,r):a2 -> A(l,x) + A(x,r) @#a1*#a2 where x = nu();
```

This way, `A` entities can form chains. Again, this can not be expressed in the reaction-based paradigm. First, a species would have to be introduced for each possible chain length, which can theoretically be infinite. Second, species linked in a chain are also still reactants for other rules, whereas for each new species representing linked entities all rules would have to be duplicated.

Finally, ML-Rules also supports arbitrary functions to encode the effect of a reaction. With "functions on solutions", which were introduced in Warnke, Helms, and Uhrmacher [239], the right side of a reaction rule can contain function calls that yield solutions. This feature can be used to encode reaction effects that are hard or impossible to express with pattern matching (see Section 4.1.4).

4.1.3. Static and dynamic compartments

A model aspect that is central to BRNs is compartmentalization [247]. By modeling a system as a set of compartments in and between which different processes occur, the natural structure of biochemical systems can be approximated. Particularly interesting are the concepts of upward causation (the contents of a compartment influence the compartment's dynamics) and downward causation (the compartment influences the dynamics of its contents). Both patterns of causal relation are common in biological systems [41]. This leads to nested systems with several levels of compartments and sub-compartments, for example a compartment with cells, where each cell contains a cell nucleus and/or different kinds of organelles (e.g., mitochondria). To make the structure of the Wnt model explicit, we can introduce the additional compartmental species `Cell` and `Nucleus`. We can also merge both β-catenin species and distinguish them regarding their location. `Bcat` is then either in the `Cell` or the `Nucleus`, and the shuttling into the nucleus (reaction 10) can be expressed with a rule.

```
Cell()[]; Nucleus()[]; // [] denotes a compartmental species
Bcat(); Wnt(); Axin(bool);

Wnt:w + Cell[Axin(true):a + s?]:c -> Wnt + Cell[Axin(false) + s?] @ k2*#c*#w*#a;

Bcat:b + Nucleus[s?] -> Nucleus[Bcat + s?] @ k10*#b;
```

[1]The name `nu()` for the function generating new unused values is inspired by the operator ν in the π-calculus playing the same role [158]. We will later also use the symbol ν for this operator in the formal definition of ML-Rules.

The `Axin(true)` in the first rule is now matched in the subsolution of the `Cell`, and the variable `s?` holds all the remaining entities in that subsolution. Therefore, `s?` is called the "rest-solution". By also using `s?` on the right side, all unmatched entities are unaffected by the reaction. In the second rule, we can omit the population size for `Nucleus`, as a cell always has exactly one nucleus. Similar to the introduction of attributes, the translation from two β-catenin species to one species located in two compartments and back is straightforward for this simple model. In more complex models, however, this can lead to much more succinct models, as one rule that can be applied in all compartments can replace many reactions, each for one compartment.

Compartmental species can also have attributes, which can then factor into the reaction rate. For example, the volume of the surrounding cell can be factored in the rate expression (downward causation) to model the degradation of cytosolic `Bcat` stimulated by phosphorylated Axin (`Axin(true)`) with correct mass action kinetics (reaction 7). As a reaction with two reactants, the rate is divided by the cell's volume (see Section 3.3):

```
Cell(num)[]; // Cell is a compartmental species with one attribute
Bcat(); Axin(bool);
Cell(vol)[Axin(true):a + Bcat + s?]:c ->
Cell(vol)[Axin(true)         + s?]   @ k7*#c*#w*#a/vol;
```

Conversely, a rule could express that the volume changes as a result of some reaction in the cell (upward causation).

Furthermore, a number of biological processes involves the creation, destruction, and modification of compartments during the simulation. In ML-Rules, dynamic compartments can be implemented naturally by having differently nested compartmental entities on the left and right rule side. For example, we can express cell division as replacing one cell with two cells [149]:

```
Cell(v)[s?]:c -> Cell(v/2)[s?] + Cell(v/2)[s?];
```

However, we have duplicated the contents of the dividing cell (the subsolution `s?`). To distribute the contents among the successor cells instead, we need the next feature of ML-Rules: functions on solutions.

4.1.4. User-defined functions and functions on solutions

An important feature of modeling languages for BRNs that is required to cope with the ever-increasing complexity of models is the ability to define and invoke functions. SBML, for example, allows for mathematical expressions to determine the rate of a reaction. However, functions are not only useful for the calculation of rates. In ML-Rules, arbitrary functions can also be used to calculate attribute values of reaction products, or to define reaction constraints based on the context in which a reaction takes place. As the following snippet shows, the syntax for function definitions is inspired by Haskell:

```
max :: num -> num -> num;          // declaration of function signature
max x y = if(x > y) then x else y; // definition
```

ML-Rules also allows "functions on solutions". These allow, for example, counting how many entities of a specific type exist in a solution without making them reactants. Functions of solutions can also return solutions. The utility of functions on solutions depends on the idea of rest-solutions, as binding a rest-solution to a variable allows using it as a function argument. For example, we can now fix the cell division rule from the previous section using functions `halfl` and `halfr`. `halfl(s?)` and `halfr(s?)` distribute the entities in `s?` into two solutions such that `halfl(s?)` + `halfr(s?)` = `s?`[2].

```
Cell(v)[s?]:c -> Cell(v/2)[halfl(s?)] + Cell(v/2)[halfr(s?)];
```

4.1.5. Deterministic events

Simulation models are typically initialized with a certain state and simulated until some stop condition (e.g., reaching a specific simulation time or a steady state). To model changes in the context of the model or extrinsic perturbations during the simulation, ML-Rules allows adding deterministic events. These are executed by the simulation algorithm at a specified time. For example, in a model of metastatic cancer, treatments such as chemotherapy can be modeled this way [20]. However, deterministic events can not be expressed with CTMC-based languages.

4.1.6. Modularization

To facilitate working with big models, some modeling languages provide constructs to factor out submodels. This is particularly helpful for recurring submodels, which can be reused multiple times after definition. Fully leveraging such modularization techniques requires that the modules can be parameterized. ML-Rules has no explicit support for model modularization and composition, but ML-Rules models have been successfully composed. For example, Haack et al. [95] composed three sub-models to investigate the interaction of intracellular and membrane kinetics in the Wnt pathway.

4.2. Other Modeling Languages for Biochemical Reaction Networks

Before and after the development of ML-Rules a plethora of other modeling languages have been developed. Each of these languages is influenced by earlier languages, and some languages are still actively developed. The individual modeling languages can mainly be distinguished by the set of features they support. We focus on the definition of reactions and rules in each language, and show snippets of the concrete syntax for illustrative purposes. All following examples have been successfully tested in the currently available language implementations.

[2]Note that this implies `halfl(s?)` ≠ `halfr(s?)` if an entity population in `s?` has an odd size.

4.2.1. Antimony

Antimony is one of the most popular tools for creating SBML-conforming models [217]. It is implemented as an external DSL with a feature set that largely matches SBML. In contrast to the XML-based SBML, however, Antimony offers a succinct, human-readable syntax. For example, a reaction of two species `L` and `R` producing an `LR` can be defined as follows:

```
L + R -> LR; k*L*R
```

The reaction rate is the product of a rate constant `k` with the population sizes of both reactant species to implement mass-action kinetics. Although, due to the limitations of SBML, Antimony currently does not support attributes and rule-based modeling, it provides some additional features. For example, the definition and reuse of modules facilitates composing models by identifying species in different modules. When exporting, modules are flattened to plain SBML and, thus, can be executed with all SBML-compatible tools.

4.2.2. BNGL

The BioNetGen Language (BNGL) is a rule-based modeling language [25]. Like other rule-based approaches, BNGL adopts a “don’t-care-don’t-write” attitude, meaning that only the aspects of the reaction rule that are necessary for its description have to be written down. For example, BNGL allows omitting the attributes of the reacting entities that do not play a role in the reaction rule. BNGL provides discrete attributes and has been designed with an emphasis on binding reactants. Similar as above, the following rule describes a reaction of two species `L` and `R` with free binding sites `r` and `l`, which are bound to each other as a result of the reaction.

```
L(r) + R(l) -> L(r!1).R(l!1) k
```

Only the rate constant `k` must be given, as the numbers of available reactants are implicitly factored into the rate (mass action kinetics). Through its extension compartmental BNGL (cBNGL), BNGL supports a fixed hierarchy of compartments that can be defined and then used in rules to constrain reactions [97]. Furthermore, the network-free simulator NFSim allows extending BNGL models with bounded integer-valued attributes [218]. However, attributes in BNGL serve primarily to encode links between entities.

4.2.3. The κ-calculus

The κ-calculus is, similarly to BNGL, a rule-based modeling language that focuses on rules that create or dissolve bindings between entities [56]. The following rule is the equivalent of the above BNGL rule in κ-calculus, where . denotes an unbound binding site:

```
L(r[.]),R(l[.]) -> L(r[1]),R(l[1]) @ 'k'
```

LBS-κ [182] is an extension of the κ-calculus that allows for modularization. Thus, models can be disassembled into reusable components. LBS-κ also enables the generation of model code through embedded F# scripts. A script with a loop, for example, can be used to generate many similar model components by generating a slightly similar string in each iteration. All strings can then be concatenated and inserted in the model code. Moreover, LBS-κ adds static compartments to the κ-calculus.

More recently, the κ-calculus has been extended with bounded counters, which allow abstracting over rules and entity states with natural-number-typed attributes [32]. Similarly to BNGL, however, the κ-calculus uses attributes primarily to encode links between entities.

4.2.4. PySB

Some modeling languages for BRNs have been defined as internal DSLs. PySB is one of the most prominent examples [142]. The language is based on the GPL Python and employs Python's programming constructs to structure the definition of models. Recurring model components can be wrapped in macros, which can then be invoked in concrete model descriptions. A sophisticated system of inheritance techniques further enables reusing models. To execute a model, PySB can communicate with BioNetGen, Kappa, or other external software. Thus, PySB is a rule-based language that adds programmatic modularization and reusability to BNGL and the κ-calculus. Individual rules can be defined in a way that resembles the syntax of those languages. This syntactical freedom is achieved by overloading operators as well as macros and other metaprogramming techniques. However, the set of available operator symbols is constrained by Python. For example, PySB uses `>>` to separate the left and right rule side, as the more common `->` is reserved by Python. The above BNGL rule

```
L(r) + R(l) -> L(r!1).R(l!1) k
```

can be encoded in PySB as

```
Rule('L_binds_R', L(r=None) + R(l=None) >> L(r=1)
```

4.2.5. Chromar

Chromar is a Haskell-based internal DSL for rule-based modeling of BRNs [106]. The language implementation is purely functional and reuses many of Haskell's features. To offer the typical rule syntax, Chromar uses quasi-quotation, a technique for macro-like compile-time code manipulation. The rule expression inside the quasi-quotes is parsed and transformed to valid Haskell code. Thus, Chromar can be characterized as an internal DSL with a sublanguage for rule definition, which is an external DSL. As an internal DSL, Haskell's features for abstraction, modularization, and reuse can be employed for Chromar models.

In contrast to BNGL and the κ-calculus, Chromar does not use attributes for linking entities, but rather to equip entities with a local state. Therefore, we use another example rule than above. The following Chromar rule expresses that two `A` entities may react by incrementing the `x` attribute of one reactant and decrementing the `x` attribute of the other. A guard is defined to prohibit that values of `x` fall below zero.

```
[rule| A{x=x}, A{x=y} --> A{x='x+1'}, A{x='y-1'} @ '1.0' ['y > 0'] |]
```

4.2.6. ℓ

The modeling language ℓ also allows for embedding GPL code in the model [256]. Here, however, the code is not used to generate model components, but rather to augment reaction rules with imperative blocks. If a rule contains such a block, the code in it is executed when a reaction derived from the rule occurs. This way, complex model behavior can be implemented imperatively as a sequence of commands in a programming language that resembles C#. The Chromar rule above can be expressed in ℓ with a `dyn` block with a conditional rate and by accessing the matched reactants directly in a `react` code block:

```
dyn [A] [A]
  when reagent_2.first(A{}).x > 1 rate 1.0
react
  var a1 := reagent_1.first(A{});
  var a2 := reagent_2.first(A{});
  a1.x := a1.x + 1;
  a2.x := a2.x - 1;
end
```

However, the available (closed-source) implementation does not contain the method `first` that is described in the corresponding paper and used to access the individual reactants. Thus, some workarounds are necessary to express a rule as the one above.

4.3. Summary

BRNs can be very effectively modeled with reaction- or rule-based languages, as reactions or rules can be defined independent of each other and models mainly are an enumeration of rules. However, different approaches to define rules exist. Various language concepts have evolved due to the individual languages' different feature sets and the different research questions they tackle. Even though the implementation strategies of the individual languages differ, the user-facing syntax and semantics are quite homogeneous. Table 4.1 shows an overview of the features of the surveyed modeling languages in comparison to ML-Rules.

One important conclusion is the central role of attributes. The expressive power of a particular language is tightly connected to its usage of attributes. For example, BNGL and the κ-calculus focus on defining binding links between entities by assigning shared attribute values to them. ML-Rules's subsolution in square brackets `[]` can be interpreted as an attribute with a solution as its value. In addition, attribute values can be used in rate expressions to constrain a reaction or influence its propensity. The expressive power

Feature	Language						
	Antimony	BNGL	κ-calculus	ML-Rules	PySB	Chromar	ℓ
Citation	[217]	[25]	[56]	[149]	[142]	[106]	[256]
Attributes							
discrete		✓	✓	✓	✓	✓	✓
continuous				✓		✓	✓
Compartments							
static	✓	✓	✓	✓	✓		✓
dynamic				✓			
User-defined functions	✓	✓	✓	✓	✓	✓	✓
Reaction effect							
Replacement	✓	✓	✓	✓	✓	✓	✓
Bindings		✓	✓	✓	✓		✓
Custom Code				✓		✓	✓
Deterministic events	✓		✓	✓		✓	
Modularization	✓		✓		✓		

Table 4.1.: Features of modeling languages for biochemical reaction networks

of attributes has been recognized and lead to the development of tailored DSLs outside the simulation community as well [5, 59].

ML-Rules distinctive feature in comparison to all other languages is its support for dynamic compartments. This is a fundamental difference, similar to the distinction of discrete and continuous attributes. First, this requires that rules need to be applied in all (sub)solutions, because it is not known which compartments will exist when running the model. Second, models with only static compartments and discrete attributes can be "flattened" by specializing the species definitions to all possible combinations of attribute values and locations. Continuous attributes, dynamic compartments, and unrestricted binding, however, makes the number of species infinite, which makes it impossible to enumerate them. Thus, any of these enables a modeling language to express models that can not be expressed with simpler languages. As ML-Rules has all of these features, it is among the most powerful modeling languages for BRNs. We will return to relating the formal expressiveness of languages in Section 5.5.4 after having discussed the formal semantics of ML-Rules.

5. Syntax and semantics of ML-Rules

We now give the abstract syntax and formal semantics of ML-Rules in an incremental fashion. Starting with a very simple version of the language, we make the language more expressive by adding new features in every step. In the end, we arrive at the full expressive power of ML-Rules. We then look into some example models and use the formalized language description as the basis for discussing ML-Rules and its distinctive features.

5.1. Population vector semantics and reaction-based modeling

For the first incarnation of ML-Rules, the semantics will be based on encoding solutions[1] as vectors. A reaction is characterized by its change vector (which can be easily determined as Products − Reactants) and its rate constant. This approach relies on models with a known finite number of populations, as described in Section 4.1.1. Later, we relax this constraint to enable the description of more complex models.

For the definitions, we rely on some prerequisites. We assume that there is a known number n of species in the model. As shown in Section 3.2, the population sizes of all species can then be represented as a vector. The set of species names is $\mathcal{S}$, and the bijective function $idx : \mathcal{S} \to \{1, \ldots, n\}$ maps each species to a unique number. For now we also assume that the change vectors of all reactions are distinct and non-zero. All bold variables, such as $\mathbf{x}$, are vectors with n elements, where the i-th element represents the size of the population of the i-th species. For example, given two species $A, B \in \mathcal{S}$ a solution with 2 A entities and 3 B entities could be denoted as $\mathbf{x} = (2, 3)$ with $idx(A) = 1$ and $idx(B) = 2$.

The abstract syntax of this first, simple language can be defined in two EBNF rules. In a reaction, reactants and products are represented as lists of species names. Each species name is preceded by a count c. When writing example rules, we do not require writing a count of 1 and just assume the count to be 1 if it is not explicitly given. Note that, for simplicity, we do not restrict usage of $\emptyset$. Thus, reactions such as $\emptyset + \emptyset + S \xrightarrow{r} \emptyset + \emptyset$ are allowed. The semantics will be responsible for handling those correctly.

[1] In the following, we will use the terms *solution* (in the biological sense) and *state* (of a CTMC) as synonyms.

$$
\begin{array}{lllll}
sol & ::= & & & \text{solution} \\
 & & cS & c \in \mathbb{N}, S \in \mathcal{S} & \text{population} \\
 & & sol + sol & & \text{addition} \\
 & & \emptyset & & \text{empty solution} \\
reac & ::= & sol \xrightarrow{r} sol & r \in \mathbb{R} & \text{reaction}
\end{array}
$$

Note that we do not define the abstract syntax of a complete model. Instead, we assume that a model consists of a set of reactions as defined by the syntax rule *reac*.

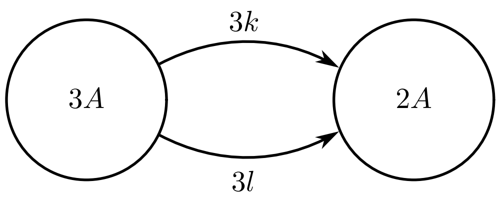
We define the operational semantics of the language with a big-step operator $\Downarrow$ that ultimately maps a model to a CTMC. The big-step operator is parameterized with a source state $\mathbf{x}$. $\Downarrow_{\mathbf{x}}$ maps each *reac* to a transition from the state $\mathbf{x}$ to a successor state $\mathbf{x}'$ with a specific propensity p, if such a transition exists. Then p is the entry $q_{\mathbf{x},\mathbf{x}'}$ of the CTMC's q-matrix. Because we assume that the change vectors of all reactions are distinct and non-zero, we can get at most one p for each $q_{\mathbf{x},\mathbf{x}'}$ with $\mathbf{x} \neq \mathbf{x}'$, and no entries $q_{\mathbf{x},\mathbf{x}}$ on the diagonal. All entries of the q-matrix for which no value is induced by the reactions of the model are assumed to be zero (or, for entries on the diagonal, the negative line sum as introduced in Section 3.1.2).

$$
\text{(EMPTY)}\frac{}{\emptyset \Downarrow_{\mathbf{x}} \mathbf{0}} \qquad \text{(NAME)}\frac{\mathbf{v} \in \mathbb{R}^n \qquad v_i = \begin{cases} c & i = idx(S) \\ 0 & otherwise \end{cases}}{cS \Downarrow_{\mathbf{x}} \mathbf{v}}
$$

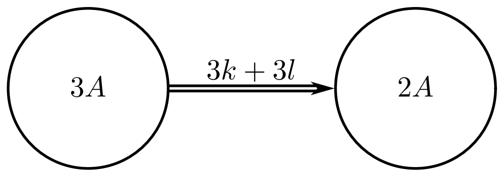
$$
\text{(SUM)}\frac{sol_1 \Downarrow_{\mathbf{x}} \mathbf{v}_1 \qquad sol_2 \Downarrow_{\mathbf{x}} \mathbf{v}_2 \qquad \mathbf{v} = \mathbf{v}_1 + \mathbf{v}_2}{sol_1 + sol_2 \Downarrow_{\mathbf{x}} \mathbf{v}}
$$

$$
\text{(REACTION)}\frac{sol_1 \Downarrow_{\mathbf{x}} \mathbf{l} \qquad sol_2 \Downarrow_{\mathbf{x}} \mathbf{k} \qquad \mathbf{l} \leq \mathbf{x} \qquad \mathbf{x}' = \mathbf{x} - \mathbf{l} + \mathbf{k} \qquad p = r \cdot \prod_{i=1}^{n} \binom{x_i}{l_i}}{sol_1 \xrightarrow{r} sol_2 \Downarrow_{\mathbf{x}} q_{\mathbf{x},\mathbf{x}'} = p}
$$

The left and the right side of a reaction are evaluated to vectors $\mathbf{l}$ and $\mathbf{k}$ that represent the entities consumed and produced by the reaction, respectively. A transition can only occur if the current state $\mathbf{x}$ contains at least the entities in $\mathbf{l}$. Then the result of evaluating the reaction is a state transition from $\mathbf{x}$ to $\mathbf{x}'$, where $\mathbf{x}'$ results from removing $\mathbf{l}$ from $\mathbf{x}$ and adding $\mathbf{k}$ instead. The propensity of this transition is determined from the number of available and needed reactants by the law of mass action kinetics (see Section 3.3). As a result, a reaction evaluates to an entry in the q-matrix of the underlying CTMC. This way, the transitions between the states of the predator-prey model as depicted in Figure 3.3 are generated from the following reactions:



(a) Each of the two reactions induces a state transition to the same successor state.



(b) Both state transitions merged by summing their propensities, yielding the unique transition between the states $3A$ and $2A$.

Figure 5.1.: Two state transitions are induced by the reactions $A \xrightarrow{k} \emptyset$ and $2A \xrightarrow{l} A$ in the state $3A$. Both lead to the successor state $2A$. The overall propensity for the transition to $2A$ is obtained by summing the propensities of the individual transitions.

$$\begin{aligned}
\text{Prey} &\xrightarrow{\lambda} 2\text{Prey} \\
\text{Predator} &\xrightarrow{\beta} \emptyset \\
\text{Predator} + \text{Prey} &\xrightarrow{\alpha} 2\ \text{Predator}
\end{aligned}$$

Before proceeding, we reconsider our assumption that the change vectors of all reactions are distinct. This assumption is quickly violated, for example with the reactions $A \xrightarrow{k} \emptyset$ and $2A \xrightarrow{l} A$. In each state with at least two A entities, both reactions induce a transition to a successor state with one less A entity. In that case the transition rate to that successor state must be the sum of both reactions' propensities (see Figure 5.1).

The issue of composing competing state transitions, some of which may lead to the same destination state, is central to defining the semantics of modeling languages with CTMC semantics [60]. There are essentially two ways of addressing this issue. First, with an appropriate encoding of state transitions, their composition can be expressed directly as shown by De Nicola et al. [58]. Roughly speaking, the approach inserts an additional layer of continuation functions in the mapping of syntax to the CTMC. By defining appropriate operations on these continuation functions, transitions can be composed elegantly. The other way of dealing with composition of state transition uses a two-step approach. It first constructs a "proto-CMTC" in which multiple transitions between a pair of state are allowed, and then in a second step sums up the transitions' rates to obtain the overall rate[2].

As we are mostly concerned with constructing the individual state transitions, we adopt the second approach. From now on, we will denote such transitions from one state $\mathbf{x}$ to

[2]Note that the SSA as presented in Section 3.1.4 can omit the second step of conflating transitions to the same destination state. It still works correctly if it samples transitions in the "proto-CTMC". Our implementations of ML-Rules will exploit this as well.

another state $\mathbf{x}'$ with a propensity of p as $\mathbf{x} \overset{p}{\Rightarrow} \mathbf{x}'$. The operational semantics will yield such transitions, and the entry in the q-matrix is defined as the sum of the propensities of all transitions from $\mathbf{x}$ to $\mathbf{x}'$. This sum is defined over a multiset (see Appendix A), as multiple reactions can yield the same propensity p. We also exploit that our model is defined as a *set* of reactions, which allows us to write statements about all reactions as multiset comprehensions with $\mathit{reac} \in \mathit{Model}$. We replace the rule (REACTION) with the following two rules:

$$\text{(REACTION)}\ \frac{\mathit{sol}_1 \Downarrow_{\mathbf{x}} \mathbf{l} \qquad \mathit{sol}_2 \Downarrow_{\mathbf{x}} \mathbf{k} \qquad \mathbf{l} \leq \mathbf{x} \qquad \mathbf{x}' = \mathbf{x} - \mathbf{l} + \mathbf{k} \qquad p = r \cdot \prod_{i=1}^{n} \binom{x_i}{l_i}}{\mathit{sol}_1 \xrightarrow{r} \mathit{sol}_2 \Downarrow_{\mathbf{x}} \mathbf{x} \overset{p}{\Rightarrow} \mathbf{x}'}$$

$$\text{(Q)}\ \frac{}{q_{\mathbf{x},\mathbf{x}'} = \sum \left\{\!\!\left| \; p \mid \mathit{reac} \Downarrow_{\mathbf{x}} \mathbf{x} \overset{p}{\Rightarrow} \mathbf{x}', \mathit{reac} \in \mathit{Model} \; \right|\!\!\right\}}$$

The premises of the rule (REACTION) are unchanged. They still express that the left side of a reaction must be a subset of the current solution and that the propensity is calculated according to mass action kinetics. The resulting state transitions are conflated by the rule (Q). In the next sections, we will omit rule (Q) and only define the CTMC up to the reactions $\mathbf{x} \overset{p}{\Rightarrow} \mathbf{x}'$ with source and destination states as well as the propensity.

5.2. Rule-based modeling: attributed species, pattern matching, and rate expressions

The next incarnation of ML-Rules now allows assigning attributes to each species. Consequently, the left sides of reactions do not necessarily refer to concrete species anymore, but can use patterns with variables, which are matched to the concrete state. This way, we avoid the combinatorial explosion caused by having to write a reaction for each combination of attribute values. Thus, our model description stays succinct by using patterns. This is the paradigm of rule-based modeling, where we specify a model with reaction rules instead of reactions (see Section 4.1.2). When applied to a concrete state, several reactions can be instantiated for each rule.

For example, consider a model of a species with a volume attribute. We can express that the entities can merge and split while preserving the overall volume in two rules:

$$\begin{aligned} A(v_1) + A(v_2) &\xrightarrow{r_m} A(v_1 + v_2) \\ A(v) &\xrightarrow{r_s \cdot v} 2A(v \div 2) \end{aligned}$$

We use variable names on the left side of the rule and determine new values for the attributes via expressions over these variables (and constants). For now, we restrict the attribute values to the reals and the expressions to standard arithmetic expressions. We will add more types and a type system later. The second rule shows that we can also use expressions to calculate the rate of the resulting reaction in dependence on the attribute values. Here we make larger As more likely to split.

The abstract syntax must be adapted to include patterns and expressions. To represent the variables, we additionally need a set of variable names $\mathcal{V}$. We additionally assume a function $ar : \mathcal{S} \to \mathbb{N}_0$ that assigns an arity to each species name. Further, for any syntactical element a, we write $\tilde{a}$ to refer to a vector $(a_1, \ldots, a_n)$, where n is clear from context.

$$
\begin{array}{llllr}
e & ::= & & & \text{expression} \\
 & & c & c \in \mathbb{R} & \text{constant} \\
 & & x & x \in \mathcal{V} & \text{variable} \\
 & & e + e \mid e - e \mid e \cdot e \mid e \div e & & \text{arithmetics} \\
sol & ::= & & & \text{solution} \\
 & & cS(\tilde{e}) & c \in \mathbb{N}, S \in \mathcal{S} & \text{entity} \\
 & & sol + sol & & \text{addition} \\
 & & \emptyset & & \text{empty solution} \\
rule & ::= & sol_1 \xrightarrow{e} sol_2 & fv(sol_2) \cup fv(e) \subseteq fv(sol_1) & \text{rule}
\end{array}
$$

We make sure that all variables that are used in the rate expression of a rule or on its right side have already been used on the left side by relating the *free variables.* $fv(\cdot)$ determines the set of all unbound variable names in a term. As we do not have a binding construct in the language yet, these are all variable names.

Introducing real-valued attributes has important consequences for the semantic foundation of the language. As there are uncountably infinitely many valuations of each species, we can no longer represent a solution with a vector of population sizes. However, we can still define a semantics that evaluates rules transitions in an underlying CTMC. To this end, we now represent solutions by multisets $\mathcal{A} = (A, n_{\mathcal{A}})$ (see Appendix A). A will be a subset of all possible entity valuations of all species: $A \subseteq \{S(\tilde{v}) | S \in \mathcal{S}, \tilde{v} \in \mathbb{R}^n, n = ar(S)\}$. For example, given a species $C \in \mathcal{S}$ with $ar(C) = 1$ a solution that consists of 3 C entities with an attribute value of 1 and one C entity with an attribute value of 2 is written as the multiset $\{\!|C(1)^3, C(2)|\!\}$, whereas the same entities would be written as $3C(1) + C(2)$ in the abstract syntax. In both cases, we allow omitting the count if it is 1.

The operational semantics is shown below. The big-step evaluator is now $\Downarrow_{\mathcal{A},\sigma}$ and parameterized with the current solution as a multiset $\mathcal{A}$ and a variable assignment $\sigma : \mathcal{V} \to \mathbb{R}$. We use the metavariables $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$ to range over multisets of entities, and $v, v_1, v_2, \ldots$ for the values of expressions, which are real numbers for now.

$$\text{(CONSTANT)}\frac{}{c \Downarrow_{\mathcal{A},\sigma} c} \qquad \text{(VAR)}\frac{\sigma(x) = v}{x \Downarrow_{\mathcal{A},\sigma} v} \qquad \text{(PLUS)}\frac{e_1 \Downarrow_{\mathcal{A},\sigma} v_1 \qquad e_2 \Downarrow_{\mathcal{A},\sigma} v_2}{e_1 + e_2 \Downarrow_{\mathcal{A},\sigma} v_1 + v_2}$$

$$\text{(ENTITY)}\frac{\begin{array}{ccc}\tilde{e} = (e_1, \ldots, e_n) & & \tilde{v} = (v_1, \ldots, v_n) \\ e_1 \Downarrow_{\mathcal{A},\sigma} v_1 & \ldots & e_n \Downarrow_{\mathcal{A},\sigma} v_n\end{array}}{cS(\tilde{e}) \Downarrow_{\mathcal{A},\sigma} \{\!\!\{ S(\tilde{v})^c \}\!\!\}}$$

$$\text{(SUM)}\frac{sol_1 \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \qquad sol_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{C}}{sol_1 + sol_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \uplus \mathcal{C}} \qquad \text{(EMPTY)}\frac{}{\emptyset \Downarrow_{\mathcal{A},\sigma} \emptyset}$$

$$\text{(RULE)}\frac{\begin{array}{c} sol_1 \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \qquad sol_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{C} \qquad e \Downarrow_{\mathcal{A},\sigma} v \qquad \mathcal{B} \subseteq \mathcal{A} \qquad \mathcal{D} = \mathcal{A} \ominus \mathcal{B} \uplus \mathcal{C} \\ \mathcal{A} = (A, n_{\mathcal{A}}) \qquad \mathcal{B} = (B, n_{\mathcal{B}}) \qquad p = v \prod_{a \in A} \binom{n_{\mathcal{A}}(a)}{n_{\mathcal{B}}(a)} \end{array}}{sol_1 \xrightarrow{e} sol_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{A} \overset{p}{\Rightarrow} \mathcal{D}}$$

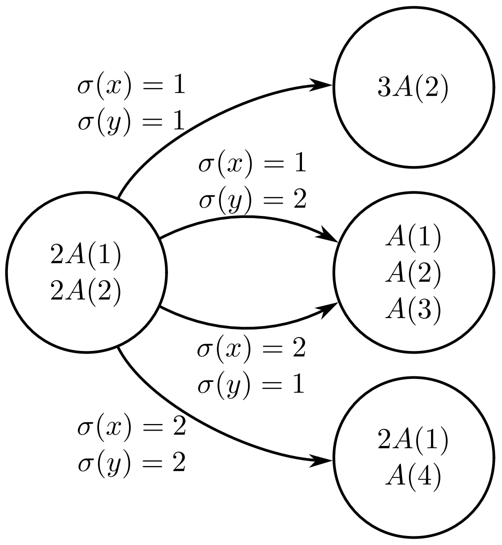
The evaluation of expressions is straightforward. Variables are read from the variable assignment σ. The assignment is never modified in any inference rule, meaning that a variable has the same value at all use sites. We give the inference rule (PLUS) as an example for defining an arithmetic operation and omit the ones for other operations.

The rule (ENTITY) allows defining entities as reactants as well as products of rules. Each attribute value is computed from an expression, which enables binding variables to values on the left rule side as well as calculating new attribute values on the right rule side.

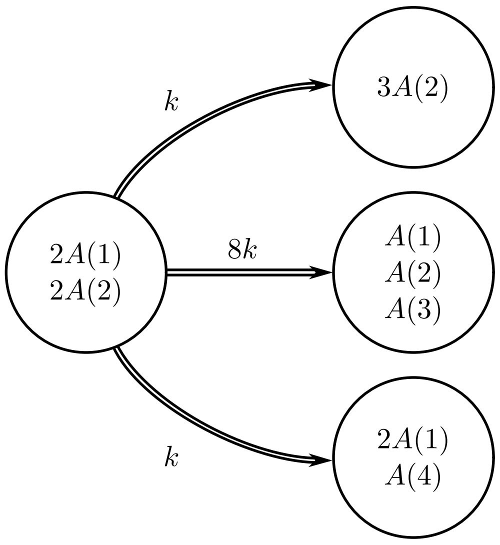
In the inference rule (RULE) we rely on the submultiset relation $\subseteq$ as well as multiset difference $\ominus$ and sum $\uplus$ (see Appendix A). Apart from the adaptations for replacing vectors with multisets, the instantiation of reactions works similarly as in the previous section. The current solution must contain at least the needed reactants. The successor state is the result of removing the reactants from the current solution and adding the products instead. To calculate the propensity, we rely on the multiplicities of the entities as encoded in the multisets.

Note that now one *rule* can induce several state transitions from a given state by instantiating it with different variable assignments σ. This is different than in the last section, where one *reac* induced at most one transition per state. Moreover, the destination states of transitions induced by one *rule* are not necessarily distinct. For example, consider the rule $A(x) + A(y) \xrightarrow{k} A(x + y)$ (Figure 5.2). If some σ induces a transition to a destination state, the same destination state is reached with the transition induced by σ' with $\sigma'(x) = \sigma(y)$ and $\sigma'(y) = \sigma(x)$. In general, however, different matches lead to different destination states[3] [106].

[3]The question of whether two matches lead to same destination states is also discussed in the Kappa manual [34]. Also see Section 3.3.2.



(a) Four different variable valuations σ lead to state transitions.



(b) The state transitions merged by summing their propensities.

Figure 5.2.: The state transitions induced by the rule $A(x)+A(y) \xrightarrow{k} A(x+y)$ in the state $2A(1) + 2A(2)$. Four state transitions lead to three successor states. The uppermost and lowermost transitions correspond to homoreactions, whereas the two center transitions correspond to heteroreactions (see Section 3.3.2). The successor state is the same if the values for x and y are flipped.

5.3. Beyond mass action kinetics: types, functions, and generalized rates

Whereas mass action kinetics are frequently used, they are by no means the only way to define reaction rates. In fact, many biochemical phenomena can only be modeled by using alternative kinetics. For example, the bistable toggle switch has the following form [103]:

$$\emptyset \xrightarrow{\frac{c_1}{c_2+(\#B)^2}} A \qquad\qquad A \xrightarrow{c_3 \cdot \#A} \emptyset$$

$$\emptyset \xrightarrow{\frac{c_4}{c_5+(\#A)^2}} B \qquad\qquad B \xrightarrow{c_6 \cdot \#B} \emptyset,$$

where the c_i are positive constants and $\#S$ is the number of entities of species S in the current state. To implement such kinetics, we allow the modeler to explicitly specify how the rate depends on the population sizes. More generally, we now allow more kinds of values and, additionally, add functions that operate on different types of values. A type system makes sure that values have the correct type for their use.

We start by defining the available types τ as the set of the base types for natural numbers, real numbers, character strings, entities, and solutions. Whereas solutions `sol` represent general multisets of entities, the type entities `ent` is used to represent one reactant of a reaction. This reactant can have a multiplicity > 1. Consequently, values of type `ent` are also multisets (with only one element with a non-zero multiplicity).

Constants now have an arity, which allows encoding functions. Thus, we introduce a set of typed n-ary constants $\mathcal{F} = \{f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau', \ldots\}$, where $ar(f) = n$ denotes the arity of f. For each $f : \tau_1 \times \cdots \times \tau_n \rightarrow \tau' \in \mathcal{F}$, there exists a function $[[f]] : \mathit{Vals}(\tau_1) \times \cdots \times \mathit{Vals}(\tau_n) \rightarrow \mathit{Vals}(\tau')$, where $\mathit{Vals}(\tau)$ denotes the set of values of type τ. The constants in $\mathcal{F}$ can encode various functions, including arithmetic operations or an if-then-else construct. Constants with an arity of zero are just typed constant values. Similarly, the set of species names includes types: $\mathcal{S} = \{S : \tilde{\tau}, \ldots\}$. Variables get assigned a type, too: $\mathcal{V} = \{x : \tau, \ldots\}$.

Expressions consist of constants that are applied to the correct number of arguments, an operator *cur* that represents the current solution (cf. John et al. [113]), and expressions to construct solutions. Consequently, the right side of a rule is now an expression. The left side of a rule is a pattern, which now also allows adding a variable to a matched entity pattern.

τ	::=			type
		`nat`		natural number
		`real`		real number
		`string`		character string
		`ent`		entity
		`sol`		solution
e	::=			expression
		$f(e_1, \ldots, e_n)$	$f \in \mathcal{F}, ar(f) = n$	application
		x	$x \in \mathcal{V}$	variable
		cur		current solution
		$eS(\tilde{e})$		entity
		$e + e$		addition
		$\emptyset_{sol}$		empty solution
pat	::=			pattern
		$cS(\tilde{e}) \triangleright x$	$c \in \mathcal{F}, ar(c) = 0, S \in \mathcal{S}, x \in \mathcal{V}$	entity pattern
		$pat + pat$		addition
		$\emptyset_{pat}$		empty pattern
$rule$	::=	$pat \xrightarrow{e_1} e_2$	$(fv(e_1) \cup fv(e_2)) \subseteq fv(p)$	rule

To allow defining rates in dependence on the reactant's population sizes, we can now define a function symbol `count` : (`ent` $\times$ `sol`) $\rightarrow$ `nat`. The expression `count`$(s, \mathcal{A})$ can be evaluated with s being an entity and $\mathcal{A}$ being a solution. It will count how often s occurs in $\mathcal{A}$. `count` ignores the multiplicity of s. If the multiplicity of s shall be taken into account, it can be factored in manually.

$$[[\mathtt{count}]](s, \mathcal{A}) = [[\mathtt{count}]](\{\!\!\{ S(\tilde{v})^v \}\!\!\}, (A, n_\mathcal{A})) = \begin{cases} n_\mathcal{A}(S(\tilde{v})) & S(\tilde{v}) \in A \\ 0 & \text{otherwise} \end{cases}$$

Now we can express the rules for the bistable toggle switch above in ML-Rules, by using cur to obtain the current solution and counting As and Bs:

$$\emptyset \xrightarrow{\frac{c_1}{c_2 + (\mathtt{count}(B, cur))^2}} A \qquad A \xrightarrow{c_3 \cdot \mathtt{count}(A, cur)} \emptyset$$

$$\emptyset \xrightarrow{\frac{c_4}{c_5 + (\mathtt{count}(A, cur))^2}} B \qquad B \xrightarrow{c_6 \cdot \mathtt{count}(B, cur)} \emptyset,$$

`count` can also be used to express mass action kinetics (see Section 3.3). This is facilitated by equipping a reactant pattern with a pattern variable, which can then be used as the first argument for `count`. For example, in Figure 5.2 we considered the rule $A(x) + A(y) \xrightarrow{k} A(x+y)$. We now express this as $A(x) \triangleright a_x + A(y) \triangleright a_y \xrightarrow{k \cdot ma} A(x+y)$, where the mass action factor ma is

$$ma = \begin{cases} \mathtt{count}(a_x, cur) \cdot \mathtt{count}(a_y, cur) & x \neq y \\ \mathtt{count}(a_x, cur) \cdot (\mathtt{count}(a_x, cur) - 1) \cdot {}^1\!/\!_2 & x = y \end{cases}$$

The first case handles heteroreactions (the center transitions in Figure 5.2), whereas the second case handles homoreactions (the uppermost and lowermost transitions in Figure 5.2). In practice, this distinction of cases can be expressed with an `if-then-else` construct comparing x and y[4]. This way, the calculation of propensities for a reaction is explicit (also see Section 3.3.2). Different approaches or interpretations of the kinetics can be encoded directly in the rate expression. For example, for every transition with $x \neq y$ there is a transition with x and y flipped leading to the same successor state. If the modeler decides that one combination of values for x and y should induce only one transition, she could replace $x \neq y$ with $x < y$ and let the rate be 0 if $x > y$.

[4]As Honorato-Zimmer et al. [106] put it, such distinctions require "a sufficiently rich stock of conditional expressions for rate expressions".

$$\text{(CONSTANT)}\;\frac{e_1 \Downarrow_{\mathcal{A},\sigma} v_1 \quad \dots \quad e_n \Downarrow_{\mathcal{A},\sigma} v_n}{f(e_1,\dots,e_n) \Downarrow_{\mathcal{A},\sigma} [[f]](v_1,\dots,v_n)} \qquad \text{(VAR)}\;\frac{\sigma(x)=v}{x \Downarrow_{\mathcal{A},\sigma} v}$$

$$\text{(CUR)}\;\frac{}{cur \Downarrow_{\mathcal{A},\sigma} \mathcal{A}} \qquad \text{(ENTITY)}\;\frac{\begin{array}{c}\tilde{e}=(e_1,\dots,e_n) \qquad \tilde{v}=(v_1,\dots,v_n)\\ e_1 \Downarrow_{\mathcal{A},\sigma} v_1 \quad \dots \quad e_n \Downarrow_{\mathcal{A},\sigma} v_n \quad e \Downarrow_{\mathcal{A},\sigma} v\end{array}}{eS(\tilde{e}) \Downarrow_{\mathcal{A},\sigma} \{\!\!\{ S(\tilde{v})^v \}\!\!\}}$$

$$\text{(SOL-SUM)}\;\frac{e_1 \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \quad e_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{C}}{e_1 + e_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \uplus \mathcal{C}} \qquad \text{(SOL-EMPTY)}\;\frac{}{\emptyset_{sol} \Downarrow_{\mathcal{A},\sigma} \emptyset}$$

$$\text{(PATTERN)}\;\frac{\begin{array}{c}\tilde{e}=(e_1,\dots,e_n) \qquad \tilde{v}=(v_1,\dots,v_n)\\ e_1 \Downarrow_{\mathcal{A},\sigma} v_1 \quad \dots \quad e_n \Downarrow_{\mathcal{A},\sigma} v_n \quad a=\{\!\!\{ S(\tilde{v})^c \}\!\!\} \quad \sigma(x)=a\end{array}}{cS(\tilde{e}) \rhd x \Downarrow_{\mathcal{A},\sigma} a}$$

$$\text{(PAT-SUM)}\;\frac{pat_1 \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \quad pat_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{C}}{pat_1 + pat_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \uplus \mathcal{C}} \qquad \text{(PAT-EMPTY)}\;\frac{}{\emptyset_{pat} \Downarrow_{\mathcal{A},\sigma} \emptyset}$$

$$\text{(RULE)}\;\frac{pat \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \quad e_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{C} \quad \mathcal{B} \subseteq \mathcal{A} \quad \mathcal{D} = \mathcal{A} \ominus \mathcal{B} \uplus \mathcal{C} \quad e_1 \Downarrow_{\mathcal{A},\sigma} v}{pat \xrightarrow{e_1} e_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{A} \overset{v}{\Rightarrow} \mathcal{D}}$$

The semantics separates terms for patterns (which only appear on the left side of the rule) from entities and solutions (which only appear on the right side of the rule). The inference rule (RULE) is again only slightly adapted from the previous version. We replaced the implicit calculation of a reaction propensity according to mass action kinetics with a propensity explicitly defined as an expression.

The new inference rule (PATTERN) requires some further explanation. It handles the evaluation of an entity pattern $cS(\tilde{e}) \rhd x$ to a concrete entity a, making sure that the variable assignment σ maps x to a. a contains the c needed copies of the reactant entity. Thus, x is also mapped to all needed copies of the reactant, and when x is added as a product of the rule, the entities a are not consumed, but remain unchanged. For example, in a rule like

$$3A(x,y,z) \rhd a + B() \rhd b \overset{\cdot}{\rightarrow} a,$$

only the B entity is consumed, and the 3 A entities are not affected.

With the introduction of functions that operate on entities and solutions, we have to make sure that the types in each rule definition line up. Therefore, we define a type system that handles each syntactical form. Most of the type constraints are straightforward. The main task of the type system is to make sure that all expressions evaluate to a value of the correct type. For example, the rate in a rule must be a real number, whereas the left and the right rule side must be solutions.

$$\text{(T-CONSTANT)}\ \frac{f : \tau_1 \times \dots \times \tau_n \to \tau' \in \mathcal{C} \qquad \vdash e_1 : \tau_1 \quad \dots \quad \vdash e_n : \tau_n}{\vdash f(e_1, \dots, e_n) : \tau'}$$

$$\text{(T-VAR)}\ \frac{x : \tau \in \mathcal{V}}{\vdash x : \tau} \qquad \text{(T-CUR)}\ \frac{}{\vdash \mathit{cur} : \texttt{sol}}$$

$$\text{(T-ENTITY)}\ \frac{\begin{array}{c} S : (\tau_1, \dots, \tau_n) \in \mathcal{S} \qquad \tilde{e} = (e_1, \dots, e_n) \\ \vdash e_1 : \tau_1 \quad \dots \quad \vdash e_n : \tau_n \quad \vdash e : \texttt{nat} \end{array}}{\vdash eS(\tilde{e}) : \texttt{ent}}$$

$$\text{(T-SOL-SUM)}\ \frac{\vdash \mathit{sol}_1 : \texttt{sol} \qquad \vdash \mathit{sol}_2 : \texttt{sol}}{\vdash \mathit{sol}_1 + \mathit{sol}_2 : \texttt{sol}} \qquad \text{(T-SOL-EMPTY)}\ \frac{}{\vdash \emptyset_{\mathit{sol}} : \texttt{sol}}$$

$$\text{(T-ENTITY-SOL)}\ \frac{\vdash \mathit{sol} : \texttt{ent}}{\vdash \mathit{sol} : \texttt{sol}}$$

$$\text{(T-PATTERN)}\ \frac{\begin{array}{c} S : (\tau_1, \dots, \tau_n) \in \mathcal{S} \qquad \tilde{e} = (e_1, \dots, e_n) \\ \vdash e_1 : \tau_1 \quad \dots \quad \vdash e_n : \tau_n \quad \vdash c : \texttt{nat} \quad \vdash x : \texttt{ent} \end{array}}{\vdash cS(\tilde{e}) \triangleright x : \texttt{ent}}$$

$$\text{(T-PAT-SUM)}\ \frac{\vdash \mathit{pat}_1 : \texttt{sol} \qquad \vdash \mathit{pat}_2 : \texttt{sol}}{\vdash \mathit{pat}_1 + \mathit{pat}_2 : \texttt{sol}} \qquad \text{(T-PAT-EMPTY)}\ \frac{}{\vdash \emptyset_{\mathit{pat}} : \texttt{sol}}$$

$$\text{(T-RULE)}\ \frac{\vdash \mathit{pat} : \texttt{sol} \qquad \vdash e_2 : \texttt{sol} \qquad \vdash e_1 : \texttt{real}}{\vdash \mathit{pat} \xrightarrow{e_1} e_2}$$

5.4. Modeling structure: compartments, rest solution, links

The last extension of ML-Rules' semantics concerns the modeling of structure and structural changes. So far, all populations are part of the same solution, that is without any further structure. We now add two structural features to the language. First, entities can now have subsolutions, which makes the model state a tree of solutions. The ability to model changes in this nesting of subsolutions, to model dynamic compartments, is ML-Rules' most distinctive feature (see Section 4.1.3). Second, entities can be linked together by sharing a unique attribute value. Both additions become new attributes or attribute types. Section 5.5.1 shows the formal definition at work in some examples.

First we look into the motivation for using links. The building of Actin filaments can be modeled by giving each Actin entity two binding sites (the barbed end and the pointed end) [218]. Then, a reaction that elongates a filament can be modeled as the creation of a new link between a free Actin and the last Actin in an existing filament. The reverse reaction severes the link between two Actins:

$$\begin{aligned} A(b, \mathit{free}) + A(\mathit{free}, p) &\xrightarrow{r_{elon}} (\nu l)\ A(b, l) + A(l, p) \\ A(b, l) + A(l, p) &\xrightarrow{r_{sev}} A(b, \mathit{free}) + A(\mathit{free}, p) \end{aligned}$$

Here, *free* is a special value of the attributes, meaning that no link value is set, and ν is an operator generating new unique link values. By only matching reactants with the value *free*, only unlinked entities are candidates for being linked. Vice versa, only entities with the same link value can be separated via the second rule.

Dynamic nesting also fits well to rule-based modeling (see Section 4.1.3). By structuring entities into nested solutions, rules only need to be written once and can then be applied to the entities in all solutions. But entities can also move between solutions and react with entities in other solutions, and the structure of solutions can change during runtime. There are many examples of biological processes where compartmental dynamics play a role. For example, we can define endocytosis (i.e., a cell "eating" some particle and wrapping it in a vesicle) by modeling cells and vesicles as entities with subsolutions:

$$\mathit{Cell}(\mathit{subsol}) + P \xrightarrow{r_{endo}} \mathit{Cell}(\mathit{subsol} + \mathit{Vesicle}(P))$$

This way, rules are able to describe the reactants' structural relation and the changes in those relations. An additional ingredient to model parts of the subsolutions that do not change is the *rest solution*. In reactant patterns, a subsolution must always be matched completely. A rule can explicitly require some entities in the subsolution and subsume the remaining entities under a variable. For example, the following rule models a particle that leaves a cell, whereas the other contents of the cell remain unchanged:

$$\mathit{Cell}(P + \mathit{rest}) \xrightarrow{r_{out}} \mathit{Cell}(\mathit{rest}) + P$$

Links and nesting require some changes to the syntax and semantics. For links, the complex part is the generation of new link names. For nesting, it is the application of rules at all structural levels.

In the abstract syntax, the operator *cur* is not meaningful anymore, as there is no single current solution[5]. Instead, we introduce the operator $\#$, which takes a variable as an argument. The expression $\#x$ will be evaluated to the count of x entities in the solution in which the pattern that the variable x is bound to has been matched. The expressions and patterns for entities have, in addition to their attribute vector, an attribute that holds their subsolution. The elements of the multisets now also have the general form $S(\tilde{v}; v)$, where v is the (potentially empty) subsolution. Patterns now contain a variable for the rest solution, and new link values for the right rule side can be generated with the ν operator. Therefore, the free variables are only the ones not bound by the ν operator. For example, in the rule

[5]There is a single root solution, which is important for representing the model state as a data structure, but not so much for the evaluation of rules.

$$A(x, \mathit{free}; s_1) + A(y, \mathit{free}; s_2) \overset{\cdot}{\rightarrow} (\nu l)A(x, l; s_1) + A(y, l; s_2)$$

the variable l is bound on the right side and, therefore, is not required to occur on the left rule side. x and y, however, are free on the right side and must occur on the left side.

τ	$::=$			type
		`nat`		natural number
		`real`		real number
		`string`		character string
		`ent`		entity
		`sol`		solution
		`link`		link
e	$::=$			expression
		$f(e_1, \ldots, e_n)$	$f \in \mathcal{F}, ar(f) = n$	application
		x	$x \in \mathcal{V}$	variable
		$\#x$		count
		$eS(\tilde{e}; e)$		entity
		$e + e$		addition
		$\emptyset_{sol}$		empty solution
pat	$::=$			pattern
		$cS(\tilde{e}; \mathit{pat}_r) \triangleright x$	$c \in \mathcal{F}, ar(c) = 0, S \in \mathcal{S}, x \in \mathcal{V}$	entity pattern
		$\mathit{pat} + \mathit{pat}$		addition
		$\emptyset_{pat}$		empty pattern
pat_r	$::=$	$\mathit{pat} + x$	$x \in \mathcal{V}$	pattern with rest
rule	$::=$	$\mathit{pat}_r \xrightarrow{e_1} (\nu\tilde{x})e_2$	$\tilde{x} \in \mathcal{V}, (\mathit{fv}(e_1) \cup \mathit{fv}((\nu\tilde{x})e_2)) \subseteq \mathit{fv}(\mathit{pat}_r)$	rule

The semantics are adapted to account for structure in the model as well.

- As the link values introduced with ν should not make a difference for the result of a rule, we introduce an equivalence relation $\equiv$ on solutions. Two solutions are equal with respect to $\equiv$ if they only differ in the generated link values. This is similar to the α-equivalence in the λ-calculus or π-calculus [159]. As usual, $[\mathcal{A}] = \{\mathcal{B} | \mathcal{A} \equiv \mathcal{B}\}$ denotes the equivalence class of $\mathcal{A}$.

- The states of the CTMC are now defined by the equivalence classes induced by $\equiv$. In the inference rule (RULE), big-step operator $\Downarrow_{\mathcal{A},\sigma}$ now associates a reaction rule with a transition $[\mathcal{A}] \overset{v}{\Rightarrow} [\mathcal{D}]$ on equivalence classes. The right rule side is evaluated with an additional valuation σ' of the link variables requested with the ν operator.

- A pattern with rest pat_r always matches the whole solution it is evaluated in. All entities not matched by the pattern are assigned to the rest solution variable, as defined in (REST). Consequently, the premises in (RULE) are simpler than before, as the left rule side now matches the complete solution.

- The state now consists of multiple nested solutions and rules can be applied at all levels. Therefore, it does not suffice anymore to only determine transitions to successor states and their propensities in the current solution. Instead, the rule (SUB) recurses into the subsolutions of all entities in a solution and "lifts" them to the current solution. In particular, if a solution $\mathcal{A}$ contains an entity $S(\tilde{v}, \mathcal{B})$ and a transition takes the subsolution $\mathcal{B}$ to $\mathcal{C}$, then a successor state of $\mathcal{A}$ is $\mathcal{A}$ with one $S(\tilde{v}, \mathcal{B})$ replaced with one $S(\tilde{v}, \mathcal{C})$ instead. If $\mathcal{A}$ contains c copies of $S(\tilde{v}, \mathcal{B})$ and the propensity of the original transition from $\mathcal{B}$ to $\mathcal{C}$ is v', then the lifted transition has a propensity of $c \cdot v'$. This way, the independent parallelism of the copies of $\mathcal{B}$ is taken into account. This lifting step can be applied multiple times, allowing to lift all transitions in all subsolutions to a single root solution. The recursion stops on empty subsolutions.

- At the same time, a single rule can already span multiple nesting levels. The inference rule (PATTERN) is changed to enable the matching of subsolution patterns. The recursion $pat_r \Downarrow_{\mathcal{B},\sigma} \mathcal{B}$ uses the subsolution of the entity as the evaluation context.

- The function # is evaluated in a similar way as variables. The variable valuation σ assigns a value to symbols of the form $\#x$, where x is a variable. In the inference rule (PATTERN) x is set to the entities actually matched by the pattern, whereas $\#x$ is set to the count of such entities in the solution. The retrieval of the value of $\#x$ is defined in the inference rule (#). Typically, expressions like $\#x$ are then used in the rate expression, for example to calculate the propensity according to mass action kinetics, replacing the `count` function from the previous section.

$$\text{(CONSTANT)}\dfrac{e_1 \Downarrow_{\mathcal{A},\sigma} v_1 \qquad \dots \qquad e_n \Downarrow_{\mathcal{A},\sigma} v_n}{f(e_1,\dots,e_n) \Downarrow_{\mathcal{A},\sigma} [[f]](v_1,\dots,v_n)}$$

$$\text{(VAR)}\dfrac{\sigma(x) = v}{x \Downarrow_{\mathcal{A},\sigma} v} \qquad\qquad (\#)\dfrac{\sigma(\#x) = v}{\#x \Downarrow_{\mathcal{A},\sigma} v}$$

$$\text{(ENTITY)}\dfrac{\begin{matrix}\tilde{e} = (e_1,\dots,e_n) & \tilde{v} = (v_1,\dots,v_n)\\ e_1 \Downarrow_{\mathcal{A},\sigma} v_1 \quad \dots \quad e_n \Downarrow_{\mathcal{A},\sigma} v_n & e \Downarrow_{\mathcal{A},\sigma} v \quad e' \Downarrow_{\mathcal{A},\sigma} \mathcal{B}\end{matrix}}{eS(\tilde{e};e') \Downarrow_{\mathcal{A},\sigma} \{\!|S(\tilde{v};\mathcal{B})^v|\!\}}$$

$$\text{(SOL-SUM)}\dfrac{e_1 \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \qquad e_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{C}}{e_1 + e_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \uplus \mathcal{C}} \qquad\qquad \text{(SOL-EMPTY)}\dfrac{}{\emptyset_{sol} \Downarrow_{\mathcal{A},\sigma} \emptyset}$$

$$\text{(PATTERN)}\dfrac{\begin{matrix}\tilde{e} = (e_1,\dots,e_n) \qquad\qquad \tilde{v} = (v_1,\dots,v_n)\\ e_1 \Downarrow_{\mathcal{A},\sigma} v_1 \quad \dots \quad e_n \Downarrow_{\mathcal{A},\sigma} v_n \quad pat_r \Downarrow_{\mathcal{B},\sigma} \mathcal{B}\\ a = \{\!|S(\tilde{v};\mathcal{B})^c|\!\} \quad \sigma(x) = a \quad \sigma(\#x) = n_{\mathcal{A}}(S(\tilde{v};\mathcal{B}))\end{matrix}}{cS(\tilde{e};pat_r) \triangleright x \Downarrow_{\mathcal{A},\sigma} a}$$

$$\text{(PAT-SUM)}\dfrac{pat_1 \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \qquad pat_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{C}}{pat_1 + pat_2 \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \uplus \mathcal{C}} \qquad\qquad \text{(PAT-EMPTY)}\dfrac{}{\emptyset_{pat} \Downarrow_{\mathcal{A},\sigma} \emptyset}$$

$$\text{(REST)}\dfrac{pat \Downarrow_{\mathcal{A},\sigma} \mathcal{B} \qquad \mathcal{A} = \mathcal{B} \uplus \mathcal{C} \qquad \sigma(x) = \mathcal{C}}{pat + x \Downarrow_{\mathcal{A},\sigma} \mathcal{A}}$$

$$\text{(RULE)}\dfrac{\begin{matrix}\tilde{x} = (x_1,\dots,x_n) \quad \sigma' : \{x_1,\dots,x_n\} \to \mathcal{L} \text{ injective}\\ pat_r \Downarrow_{\mathcal{A},\sigma} \mathcal{A} \qquad e_2 \Downarrow_{\mathcal{A},\sigma\cup\sigma'} \mathcal{D} \qquad e_1 \Downarrow_{\mathcal{A},\sigma} v\end{matrix}}{pat_r \xrightarrow{e_1} (\nu\tilde{x})e_2 \Downarrow_{\mathcal{A},\sigma} [\mathcal{A}] \overset{v}{\Longrightarrow} [\mathcal{D}]}$$

$$\text{(SUB)}\dfrac{\begin{matrix}S(\tilde{v},\mathcal{B}) \in \mathcal{A} \qquad rule \Downarrow_{\mathcal{B},\sigma} [\mathcal{B}] \overset{v'}{\Longrightarrow} [\mathcal{C}] \qquad \mathcal{D} = \mathcal{A} \ominus \{\!|S(\tilde{v},\mathcal{B})|\!\} \uplus \{\!|S(\tilde{v},\mathcal{C})|\!\}\\ v = v' \cdot n_{\mathcal{A}}(S(\tilde{v},\mathcal{B}))\end{matrix}}{rule \Downarrow_{\mathcal{A},\sigma} [\mathcal{A}] \overset{v}{\Longrightarrow} [\mathcal{D}]}$$

We omit the type system for this last incarnation of ML-Rules, as it is largely equal to the previous one. The most prominent change is the introduction of the type `link` and the constraint that the subsolution attribute must be of type `sol`.

This concludes the definition of the abstract syntax and formal semantics of ML-Rules. ML-Rules' most distinctive feature, the modeling of dynamically nested multisets of attributes entities, is captured in the evaluation rules of this last incarnation of the formal language definition. In particular the inference rules (PATTERN) and (SUB) characterize ML-Rules. (PATTERN) handles patterns with nested entities on the left rule side, whereas (SUB) makes sure that rules are applied at all nesting levels. This way, dynamic nesting can be succinctly expressed with reaction rules.

5.5. Discussion

The formal definition of both syntax and semantics allows us to discuss some implementation-independent properties of ML-Rules.

5.5.1. Example models in the abstract syntax

To illustrate the abstract syntax, we show how it can be employed to encode models of BRNs. As the abstract syntax focuses on the reaction rules, we only show the rules of the models. These rules contain references to species and constants, which will be clear from context. We point out interesting implications regarding the representation of the underlying models as well as interdependencies between the syntax and semantics. As a syntactical shortcut, we omit the rest solution at the root level of a rule and the parentheses of entities without attributes (cf. [176]). We also write solutions in ML-Rules' abstract syntax instead of writing them explicitly as multisets.

Wnt model

We first take a look at how the Wnt model introduced in Section 4.1 can be expressed in the abstract syntax. To illustrate how the formal semantics describes how the rules evaluate to reactions, we look at a few rules and how they are applied to a concrete solution. The rules 2, 7, and 10 were the ones we already saw in ML-Rules' concrete syntax. Rule 2 models the dephosphorylation of Axin stimulated by extracellular Wnt, rule 7 models the degradation of β-catenin stimulated by phosphorylated Axin, and rule 10 models the shuttling of β-catenin into the nucleus. These rules can be expressed in the abstract syntax as follows:

$$Wnt \triangleright w + Cell(vol; Axin(true) \triangleright a + s) \triangleright c \xrightarrow{k_2 \cdot \#w \cdot \#a \cdot \#c} Wnt + Cell(vol; Axin(false) + s) \quad (2)$$

$$Cell(vol; Axin(true) \triangleright a + Bcat \triangleright b + s) \triangleright c \xrightarrow{k_7 \cdot \#a \cdot \#b \cdot \#c \div vol} Cell(vol; Axin(true) + s) \quad (7)$$

$$Bcat \triangleright b + Nucleus(; s) \xrightarrow{k_{10} \cdot \#b} Nucleus(; Bcat + s) \quad (10)$$

Based on these three rules, we can use the formal semantics to determine what reactions can occur in a given solution with what reaction rates. As an example, we consider the following solution:

$$\begin{aligned}1000\, Wnt + Cell(2;\ & 200\, Axin(true)+\\ & 250\, Axin(false)+\\ & 12000\, Bcat+\\ & Nucleus(;5000\, Bcat))\end{aligned}$$

For each rule, there is one way to match the left rule side to the solution; rule 10 can only be matched in the subsolution of the cell. Thus, there are the following reactions:

- Rule 2 matches with the variable assignment $\sigma = \{$

  $$\begin{aligned}w &\mapsto Wnt,\\ \#w &\mapsto 1000,\\ vol &\mapsto 2,\\ a &\mapsto Axin(true),\\ \#a &\mapsto 200,\\ s &\mapsto 199\, Axin(true) + 250\, Axin(false) + 12000\, Bcat + Nucleus(;\ 5000\, Bcat),\\ c &\mapsto Cell(2; 200\, Axin(true) + 250\, Axin(false) + 12000\, Bcat + Nucleus(;\ 5000\, Bcat)),\\ \#c &\mapsto 1\}\end{aligned}$$

  The successor state is

  $$\begin{aligned}1000\, Wnt + Cell(2; & 199\, Axin(true)+\\ & 251\, Axin(false)+\\ & 12000\, Bcat+\\ & Nucleus(;\ 5000\, Bcat))\end{aligned}$$

  and the rate is $k_2 \cdot \#w \cdot \#a \cdot \#c = 200000k_2$.

- Rule 7 matches with the variable assignment $\sigma = \{$

  $$\begin{aligned}vol &\mapsto 2,\\ a &\mapsto Axin(true),\\ \#a &\mapsto 200,\\ b &\mapsto Bcat,\\ \#b &\mapsto 12000,\\ s &\mapsto 199\, Axin(true) + 250\, Axin(false) + 11999\, Bcat + Nucleus(;\ 5000\, Bcat),\\ c &\mapsto Cell(2; 200\, Axin(true) + 250\, Axin(false) + 12000\, Bcat + Nucleus(;\ 5000\, Bcat)),\\ \#c &\mapsto 1\}\end{aligned}$$

  The successor state is

  $$\begin{aligned}1000\, Wnt + Cell(2; & 200\, Axin(true)+\\ & 250\, Axin(false)+\\ & 11999\, Bcat+\\ & Nucleus(;5000\, Bcat))\end{aligned}$$

and the rate is $k_7 \cdot \#a \cdot \#b \cdot \#c \div vol = 1200000k_7$.

- Rule 10 matches with the variable assignment $\sigma = \{$

  $$\begin{aligned} b &\mapsto Bcat, \\ \#b &\mapsto 12000, \\ s &\mapsto 5000\, Bcat\}. \end{aligned}$$

  The successor state is

  $$\begin{aligned} 1000\, Wnt + Cell(2;\, &200\, Axin(true)+ \\ &250\, Axin(false)+ \\ &11999\, Bcat+ \\ &Nucleus(;5001\, Bcat)) \end{aligned}$$

  and the rate is $k_{10} \cdot \#b = 12000k_{10}$.

The successor states regarding the remaining rules can be determined similarly. As this model is an adaptation of a reaction-based model, each rule leads to at most one reaction for the given solution with only one cell. However, the impact of the rule-based approach with attributed compartments becomes evident if the solution contains several cells with distinct volumes or subsolutions. Then each rule produces a match for each cell. A corresponding experiment with a population of 100 cells has been described in the original publication by Mazemondet et al. [150].

Cell cycle model

As a second example, we express in the abstract syntax a simplified version of Tyson's cell cycle model as adapted by Maus et al. [230, 149]. The model includes the molecular species and intracellular processes that lead to DNA replication and mitosis. The enzyme maturation promoting factor (MPF) is produced in an inactive state by a reaction of cyclin (Y) and cdc2 (D). Whereas cyclin is synthesized in the cell, we assume a fixed number of cdc2 per cell. Inactive MPF (M_I) is then activated in an autocatalytic reaction, yielding active MPF (M_A). Active MPF breaks to cdc2 (which can again be used to build inactive MPF) and phosphorylated cyclin Y_P (which degrades rapidly). These five intracellular reactions can be visualized as a network of reactions (Figure 5.3).

These reactions occur inside a cell (C) whose volume influences the reaction rates. Conversely, the cell goes through several phases that are controlled by the intracellular state. We model the volume and phase as attributes of the cell and define reactions on the cell that depend on the counts of the species in the cell. We distinguish three phases $G1$, $SG2$, and M, and four processes:

- A cell in phase $G1$ or $SG2$ can grow (i.e., increase its volume)
- A cell changes from phase $G1$ to $SG2$ if it contains more M_I than a threshold T_7

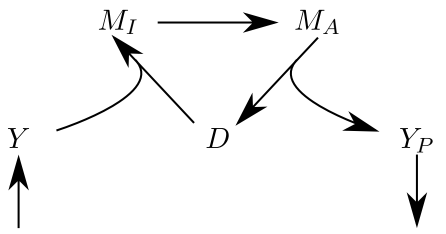


Figure 5.3.: A visualization of the five intracellular reactions of the simple cell cycle model.

- A cell changes from phase $SG2$ to M if it contains more M_A than a threshold T_8
- A cell divides if it is in phase M and contains less M_A than a threshold T_9

In this model, intracellular processes depend on the cell's volume and the cell growth and phase transitions depend on the intracellular processes. This shows the importance of representing dynamic compartments and multiple levels of organization in the modeling language. The third rule exemplifies how some reaction rates do not follow mass action kinetics, in this case to model an autocatalytic process.

$$\begin{array}{rl}
C(v,p;s) \triangleright c \xrightarrow{k_1 \cdot v \cdot \#c} & C(v,p;Y+s) \\
C(v,p;Y \triangleright y + D \triangleright d + s) \triangleright c \xrightarrow{k_2 \cdot \#c \cdot \#y \cdot \#d \cdot 1/v} & C(v,p;M_I + s) \\
C(v,p;M_I \triangleright i + s) \triangleright c \xrightarrow{(k_3' + (k_3 \cdot (count(M_A,\, s)/D_{tot})^2)) \cdot \#i \cdot \#c} & C(v,p;M_A + s) \\
C(v,p;M_A \triangleright a + s) \triangleright c \xrightarrow{k_4 \cdot \#a \cdot \#c} & C(v,p;Y_P + D + s) \\
Y_P \triangleright y \xrightarrow{k_5 \cdot \#y} & \emptyset \\
C(v,p;s) \triangleright c \xrightarrow{\texttt{if } p \in \{G1, SG2\} \texttt{ then } k_6 \cdot \#c \texttt{ else } 0} & C(v + 1/T_d, p; s) \\
C(v,G1;s) \triangleright c \xrightarrow{\texttt{if } count(M_I,s) > T_7 \texttt{ then } k_7 \cdot \#c \texttt{ else } 0} & C(v,SG2;s) \\
C(v,SG2;s) \triangleright c \xrightarrow{\texttt{if } count(M_A,s) > T_8 \texttt{ then } k_8 \cdot \#c \texttt{ else } 0} & C(v,M;s) \\
C(v,M;s) \triangleright c \xrightarrow{\texttt{if } count(M_A,s) < T_9 \texttt{ then } k_9 \cdot \#c \texttt{ else } 0} & C(v/2, G1; \mathit{fill}(\mathit{half}_l(s))) + \\
 & C(v/2, G1; \mathit{fill}(\mathit{half}_r(s)))
\end{array}$$

The functions half_l and half_r distribute the contents of a cell into two daughter cells without creating or losing any entities. To restore the total number of contained entities per cell, the function *fill* fills up the D population to maintain the sum of $\#D + \#M_I + \#M_A = D_{tot}$ in both daughter cells.

$$\begin{aligned}
[[\mathit{half}_l]]((A, n_{\mathcal{A}})) &= (A, n_{\mathcal{B}}), \text{ where } n_{\mathcal{B}}(x) = \lfloor n_{\mathcal{A}}(x)/2 \rfloor \text{ for all } x \in A \\
[[\mathit{half}_r]]((A, n_{\mathcal{A}})) &= (A, n_{\mathcal{B}}), \text{ where } n_{\mathcal{B}}(x) = \lceil n_{\mathcal{A}}(x)/2 \rceil \text{ for all } x \in A \\
[[\mathit{fill}]](\mathcal{A}) &= \mathcal{A} \uplus \{\!\!\{ D^{D_{tot} - D_{old}} \}\!\!\}, \text{ where } D_{old} = n_{\mathcal{A}}(M_A) + n_{\mathcal{A}}(M_I) + n_{\mathcal{A}}(D)
\end{aligned}$$

Deeply nested models

We illustrate the effect of nesting on reaction rates in a final example. Consider the following, deeply nested solution:

$$10\,A(;10\,A(;B))$$

The solution contains 10 A entities, each of which contains 10 A entities, each of which contains 1 B entity. Thus, the solution contains 110 A and 100 B entities. Now we define a rule that has one B entity as its sole reactant:

$$B \xrightarrow{k} 2B$$

This rule only matches once, in the innermost subsolution. However, when the corresponding reaction occurs, it occurs in only 1 A entity. The inference rule (SUB) lifts the reaction occurring in the inner A entity's subsolution to the root solution. The resulting successor state is

$$9\,A(;10\,A(;B)) + A(;9\,A(;B) + A(;2B))$$

and the rate for the reaction is $10 \cdot 10 \cdot k = 100k$.

5.5.2. Provability of properties and static analysis

One direct consequence of having formal definitions of syntax and semantics is the ability to prove properties of the language. For example, it is possible to prove that two terms (expressed in the abstract syntax) are semantically equivalent. This should not be underestimated, as it allows simulation algorithms to rewrite a model before executing it without changing its semantics, similarly to optimizing compilers [17]. For example, consider a rule of the following form

$$S(\tilde{v};v) \triangleright s + S(\tilde{v};v) + \ldots + S(\tilde{v};v) \xrightarrow{f(\#s)} e,$$

where S is some arbitrary species name, $\tilde{v}$ and v are values, f is a function that calculates a rate, and e is some expression that does not use s. We assume that all expressions are typed correctly.

Let the number of reactants $S(\tilde{v};v)$ be n. Then it is equivalent to write the rule as

$$nS(\tilde{v};v) \triangleright s \xrightarrow{f(\#s)} e.$$

The equivalence can be shown by evaluating both variants of the rule with the big-step evaluator $\Downarrow_{\mathcal{A},\sigma}$ and the same $\mathcal{A}$ and σ. Evaluating one variant leads to the multiset $\{\!|S(\tilde{v};v)^n|\!\}$ if and only if evaluating the other variant leads to the same multiset.

In a similar fashion, the irrelevance of the reactant's ordering can be shown by considering the inference rule (PAT-SUM) in the formal semantics. The terms $p_1 + p_2$ and $p_2 + p_1$ are equivalent due to the commutativity of the (implicit) logical conjunction in the rule's premise as well as the multiset sum on the right side of the big-step evaluator.

In addition, the formal semantics allows analyzing models statically without executing them (see Section 2.4.2). Applications include reachability analysis, which answers the question whether a specific entity (a species with specific values for each attribute) can be produced during a simulation run. The Kappa Static Analyzer [35] demonstrates the feasibility of this approach for the κ-calculus. Static analyses in the context of ML-Rules are discussed further in Section 6.5.3.

5.5.3. Practical issues

The definition of the abstract syntax and formal semantics of ML-Rules happens on a very abstract level. However, it already allows some observations about the implementation and use of ML-Rules in practice. Some of these aspects will reappear in Chapter 6.

- Most obviously, the users of the language use a concrete syntax rather than the abstract syntax. This allows various syntactic shortcuts. We have already started to make the abstract syntax more readable by omitting the rest solution at the highest hierarchy level in a rule. The abstract syntax and formal semantics, however, are simpler to define when they are as explicit as possible.
- By equipping a species with attributes and subsolutions, it becomes less probable that two entities are equal and can be summarized in a population. Thus, simulation algorithms might treat some species as individuals instead of populations. See Chapter 6.6 for a more thorough discussion on this.
- The underlying CTMC might be very hard to handle with analysis tools, as, for example, the states become very complex with attributed and nested species. However, it is still a CTMC and the formal semantics unambiguously specifies what transitions are possible from a specific state. Any simulation algorithm can use this as guidance. In addition, static analysis of the model (the rules, not the state space) might reveal that the model does not exploit all possible features of the language (see Section 6.5.3). For such models with reduced feature sets, specialized, more efficient algorithms can be developed [101].
- As long as it fits into the type system, any kind of expression language can be used. An alternative to the n-ary constant functions that we used would be the lambda-calculus. In practice, some commonly needed functions can be predefined, and the modeler can define additional functions in some arbitrary syntax. Similarly, the set of available types can be extended as needed. A `let-in` or `where` construct [130] construct might be helpful, for example to implement the ν-operator. ML-Rules 2 features `where` expressions (see Section 4.1.2).

Two different implementations of ML-Rules existed already before the abstract syntax and formal semantics was defined. Internally, these made different decisions about how to implement ML-Rules, which was only informally defined then. Formal definitions allow assessing the faithfulness of specific implementation variants. However, more thorough considerations about simulator implementation strategies, algorithmic optimizations, or simulator performance characteristics are not the focus of this thesis.

5.5.4. Comparison with other formally defined languages

ML-Rules is not the first modeling language for biochemical reaction networks for which formal syntax and semantics have been defined. There are other languages that influenced the development of ML-Rules and can be related to it in terms of formal expressiveness.

React(C)

The foundation of formal definitions for modeling languages for biochemical reaction networks is the π-calculus [159] and its descendant, the stochastic π-calculus [192]. The stochastic π-calculus, in turn, gave rise to the attributed π-calculus [114] and then the language *React*(C) [113]. There exist a number of proofs that relate these languages to each other. These proofs mostly work by showing that every construct of the source language can be translated to a construct in the target language while retaining the semantics. This way, it can be shown that the source language is not more expressive than the target language. For example, John et al. [113] have defined a formal semantics for *React*(C) and used it to show that the stochastic π-calculus can be expressed in *React*(C). Thus, the stochastic π-calculus is at most as expressive as *React*(C), but not more expressive.

React(C) as well as ML-Rules are successors of the attributed π-calculus [114]. The attributed π-calculus extends the stochastic π-calculus with process attributes, which can be used to constrain synchronization between processes or calculate the stochastic rate of synchronization. Due to their common ancestry, *React*(C) and ML-Rules share many concepts in syntax and semantics. Consequently, the formal semantics of *React*(C) was a significant influence for the definition of ML-Rules' formal semantics.

We exploit the similarities of *React*(C) and ML-Rules by relating both to each other and extending the known and proven properties of *React*(C) to ML-Rules. However, we do not give formal proofs here. Instead, we shortly discuss some commonalities and differences of both languages.

- First, ML-Rules and *React*(C) use different approaches for formulating expressions, for example to count entities for reaction rates. Both allow arbitrary constant functions. *React*(C) additionally includes the λ-calculus including recursion via a `letrec` operator and combines primitive functions such as $+$ (addition of numbers) or $=$ (comparison of entities). This way, counting entities can be implemented directly in the syntax of *React*(C). In contrast, ML-Rules has a built-in function # for counting entities. Although the syntax of ML-Rules does not allow as many functions as *React*(C), we can still describe the semantics of arbitrary function constants. In practice, commonly needed function would be better provided by the language implementation (or at least in some kind of standard library) instead of being implemented as part of a model by language users. Allowing users to define their own functions can be considered an orthogonal issue.

- Compartments are supported by ML-Rules and *React*(C) in different ways. In *React*(C), an entity's compartment is denoted by an attribute of the entity. Thus, a reaction can be constrained to entities in the same compartment by requiring that all reactant share a common compartment attribute. In ML-Rules, compartmental entities have an attribute that holds the subsolution, the multiset of the entities in the compartment. Note that ML-Rules also enables modeling compartments with attributes and links, similar to *React*(C). However, modeling compartments explicitly is arguably preferable. For example, in ML-Rules by default only entities localized in the same compartment can react, unless a rule specifies how the reaction crosses compartment borders. This prohibits unrealistic long-range interactions.

- Both language support models in which compartments can be nested infinitely. In ML-Rules, a rule $A(;s) \xrightarrow{r} A(;s + A(;\emptyset_s))$ allows each A entity to create another A entity in its subsolution. In *React*(C), an equivalent rule is $A(c,p) \xrightarrow{r} (\nu\ c_2)A(c,p), A(c_2,c)$, with A's attributes standing for hyperedges to *c*hildren and to the *p*arent.

- A result of the different handling of compartments is that ML-Rules allows abstracting over the entities in a compartment, whereas the compartment attribute in *React*(C) is species-specific. For example, in ML-Rules we can obtain the number of entities in a subsolution of an entity $A(;sol)$ with a function call like $size(sol)$. In contrast, *React*(C) requires summing the count of all species populations to obtain the number of entities that share a compartment attribute x: $count\ A(x) + count\ B(x) + \ldots$. Thus, such a function can not be implemented in a model-agnostic fashion.

Based on these informal considerations, we can not conclude whether there are *React*(C) models which can not be expressed with ML-Rules or vice versa. However, it is clear that the important difference is in the representation of compartments. *React*(C) relies on an implicit encoding of compartments, and ML-Rules makes them explicit, while also allowing, but not encouraging, an implicit representation via links. Thus, ML-Rules has a strictly larger vocabulary, which is a hint that it is formally more expressive. On the other hand, there is no obvious example for an ML-Rules model that can not be expressed in *React*(C), which hints at ML-Rules being not more expressive than *React*(C). Independently of that, ML-Rules allows for a higher degree of abstraction, as the set of model-independent functions that can be defined in ML-Rules is a strict superset of the set of functions possible in *React*(C).

Term rewriting

In addition to the mental framework based on process algebras, rule-based modeling can also be considered from a term rewriting perspective. To illustrate the connection, we review the work by Oury and Plotkin on "(coloured) stochastic multilevel multiset rewriting " ((C)SMMR) [175, 176].

Similar as in ML-Rules, solutions are represented as nested multisets. In contrast to the approaches considered so far, however, the multisets are represented as algebraic terms for which specific laws hold. In particular, the addition of multisets is associative, commutative, and has the neutral element $\emptyset$ (the empty set). These laws can be expressed as the equivalence relation $\approx_{AC1}$:

$$\begin{aligned}(x+y)+z &\approx_{AC1} x+(y+z)\\ x+y &\approx_{AC1} y+x\\ x+\emptyset &\approx_{AC1} x\end{aligned}$$

for any multiset x, y, and z[6]. Oury and Plotkin use a comma symbol , instead of a plus symbol + to denote multiset addition.

The multisets are made up of atomic terms, which represent biochemical species. Constants represent species without a subsolution, and unary function symbols represent compartmental species (with the single argument representing the subsolution). In addition, constants and function symbols are parameterized with labeled (color) values. For example, a cell C with a volume of 1 containing 10 A and 5 B entities could be expressed as

$$C(1; 10A() + 5B())$$

in ML-Rules' abstract syntax. For comparison, in CSMMR we would write the term

$$C_{vol:1}(10A(), 5B()).$$

Given the identification of solutions with algebraic terms, a reaction rule can be expressed as rewriting a term. This allows reusing the research on term rewriting for modeling biochemical reaction networks. In particular, the matching of reactant patterns and binding of variables on the left rule side relies on established term rewriting methods [10, pp. 262f]. Through appropriate definitions, this can be extended to calculating rates for state transitions in a CTMC, similar to our formal semantics for ML-Rules. To continue the example, an elimination reaction of an A and a B in a cell as shown above, which depends on the cells volume, would be expressed in ML-Rules' abstract syntax as

$$C(vol; A() \triangleright a + B() \triangleright b + s) \triangleright c \xrightarrow{\#a \cdot \#b \cdot \#c \cdot k / vol} C(vol; s),$$

whereas in CSMMR we would write

$$C_{vol:v}(A(), B(), s) \xrightarrow{k/v} C_{vol:v}(s).$$

Both languages are quite similar syntactically, but there are some differences. ML-Rules requires explicitly defining the reaction rate in dependence of population sizes, whereas CSMMR assumes mass action kinetics by implicitly counting occurrences of the rule in a given solution. Another difference is that ML-Rules identifies attributes by their

[6]This justifies that we never write parentheses and freely permute the ordering of solutions and patterns.

position, while CSMMR assigns labels to attributes. However, both languages also have much in common. Most importantly, both languages support an explicit representation of dynamic compartments. By separating attributes and subsolution syntactically, the significance of the nesting structure is emphasized. Functions on attributes can be easily integrated in CSMMR due to its relation to algebra, but functions on solutions, which are also supported in ML-Rules, have not been included so far.

In summary, it is interesting to see two languages with similar syntax and feature sets defined on so different bases. The area of term rewriting offers many concepts that are reusable for defining a rule-based modeling language for biochemical reaction networks. As Oury and Plotkin put it, "by working within term rewriting, with its algebraic setting, one employs a very standard approach; this contrasts with other rule-based approaches which, while perfectly sound, are, perhaps, somewhat *ad hoc*" [176]. However, it seems as if the idea was not further pursued, and no implementation of the language has been published. Nevertheless, in Section 6.2.1 we will be able to use some insights from term rewriting for implementing ML-Rules although ML-Rules' formal definition does not employ term rewriting.

The κ-calculus and BNGL

The BioNetGen language (BNGL) [25] and the κ-calculus [56], which were already shortly introduced in the sections 4.2.2 and 4.2.3, respectively, are arguably the most mature rule-based modeling languages for biochemical reaction networks. Both languages focus on relatively simple entities that form complexes by building links between the entities' binding sites (see also Section 4.1.2). As a consequence, complexes are represented as typed, attributed graphs, and reaction rules are defined using pattern graphs. The applicability of a reaction rule can be determined by finding a sub-graph isomorphism, and applying a reaction corresponds to graph rewriting [56, 26]. This way, biochemical reaction networks can be mapped to well-known concepts.

Similar to ML-Rules's semantics, the semantics of both BNGL and the κ-calculus mostly revolve around pattern matching and reaction instantiation. As mentioned above, pattern matching is reduced to finding a sub-graph isomorphism and reactions to graph rewriting. Having defined the reaction instantiation, the definition of a CTMC and, thus, a simulation algorithm for the language is straightforward. In contrast to ML-Rules, however, the languages' foundation in graph theory imposes some additional conditions on what models can be expressed and how. We give a few examples here.

First, patterns in BNGL and the κ-calculus adopt more of a "don't-care-don't-write" attitude than ML-Rules, as they allow to omit all unchanged and noninfluential attributes and links of the reactants. As a consequence, both languages require in each rule a mapping between reactants and products. This mapping allows transferring the unchanged values from matched reactants to the corresponding products. Whereas the mapping can be often inferred automatically, both BNGL and the κ-calculus allow explicitly mapping reactants to products [73]. Such explicitness is particularly useful when a reaction removes or creates an entity, as then the numbers of reactants and products are not equal. For example, the following rule in the κ-calculus specifies a "hole" . on the right rule side to denote that the first reactant is removed [34]:

```
L(r[1]), R(l[1]) -> ., R(l[.]) @ 'k'
```

The foundation in graph theory also means that BNGL and the κ-calculus only allow binary links between entities (or complexes). This excludes hypergraphs, which could be used to represent compartmental structures[7]. Extensions to both BNGL and the κ-calculus have been proposed to include compartments [97, 182]. However, these extensions are somewhat orthogonal to the original language concepts and, thus, do not allow expressing changes in the compartments with reaction rules. In contrast, ML-Rules allows expressing changes in the nesting by specifying differently nested entities on the left and right rule side. Similarly, abstraction over compartments is limited in comparison to ML-Rules While ML-Rules allows constraining rules to only be applied in specific compartments, it also allows writing rules that are independent of their context, if desired[8]. This way, rules do not need to refer to concrete compartments, facilitating dynamic compartments, which are not supported in BNGL or the κ-calculus.

Finally, the graph interpretation leads to some corner cases that need to be handled in the languages. For example, BNGL allows augmenting reaction rules with a number of additional directives, such as the keyword `DeleteMolecules`, which controls how the removal of a single entity affects other entities linked to it [71]. Faeder, Blinov, and Hlavacek [71, pp. 135f.] state:

> These commands have been introduced to address the need for specific behaviors that are difficult or impossible to specify using the semantics of patterns and transformation rules alone. [...] In the future, we anticipate the development of a “pattern logic” that will provide these capabilities in a more general way.

Similarly, corner cases need to be considered when determining the rate for a reaction instance based on the pattern matching, not unlike the discussion in Section 3.3.1. For example, the Kappa manual includes a discussion of so-called “ambiguous molecularity” that occurs when a reaction rule can lead to a reaction of two complexes and also to a reaction of a complex with itself [34]. As these correspond to second-order and first-order reactions, respectively, the surrounding volume factors differently into the rate expression. In addition, symmetries in the pattern matching may occur, as shortly discussed in Section 3.3.2. ML-Rules, in contrast, avoids such corner cases by making no assumptions about the kinetics and requiring rate expressions to explicitly factor in, for example, population sizes or compartment volumes.

[7]In fact, one of the motivations for developing *React*(C) was extending the κ-calculus with hyperedges [113].

[8]Compare the left rule sides $C(;A+B+s)$ and $A+B$, for example. The first rule can only induce reactions inside a C, whereas the second rule applies everywhere. In addition, the first rule can be further constrained using the attributes of C.

5.6. Summary

The formal definition of the syntax and semantics of ML-Rules is one of the core aspects of this thesis. In the iterative development of the language, we have seen that higher expressiveness is bought with more complex semantics rules. Whereas the first iteration only supported reaction-based modeling, we added attributes, pattern matching, and rule-based modeling for the second version. The next step was to make expressions more powerful, for instance by adding functions. Functions can be applied to solutions as well, which allows for user-defined rate expressions. This made a type system necessary, for example to ensure that user-defined rate expressions actually yield a numeric value. As a final step, we included dynamic compartments. This lead to an additional semantics rule for determining all state transitions. The reason is that with a compartmental structure that develops while the model is running, reaction rules need to be applied in all compartments.

Over all these levels, the formal semantics specifies unambiguously how the syntactic representation of an ML-Rules model gives rise to a CTMC. To do so, it suffices to define the semantics of the reaction rules and just assume an appropriate state space as given. As we will see in the next chapter, the reaction rules also dominate the practical implementation of ML-Rules.

Besides guiding the implementation, the formal definition of ML-Rules also allows some considerations on the abstract level. We were able to define example models in the abstract syntax and discuss the features of ML-Rules independently of a specific implementation. The formal definition also enables relating model fragments and determining that, for example, two formulations of a rule are equivalent. Such deliberations can be valuable when implementing simulation algorithms, as it allows rewriting a model fragment into a semantically equivalent, but more computationally efficient version.

Finally, the formal definition allows reasoning about the relation between modeling languages. For example, we were able to compare ML-Rules and $React(C)$ in terms of expressiveness by looking beyond the syntax (which is quite different) and seeing how dynamic compartments, an advanced feature, can be realized in both languages. On the other hand, we were able to see how ML-Rules relates to CSMMR, which is based on term rewriting, as well as the κ-calculus and BNGL, which are based on graph rewriting. These two semantic foundations provide specific perspectives for expressing patterns (on the left side of reaction rules) and matching them to a concrete solution. In particular, term rewriting favors tree-like structures (corresponding to compartments), whereas graph rewriting leads to network structures (corresponding to links). Both areas also offer algorithms for pattern matching and will continue to provide theoretical grounding for rule-based modeling languages.

In summary, formally defining the syntax and semantics of a modeling language is an important aspect of unlocking the potential of using modeling languages in the first place. It provides the right level of abstraction to expose the idiosyncrasies of a language. These idiosyncrasies, in turn, give impulses for the further development of modeling languages. More immediately, they inform the implementation of the language, which is the topic of the next chapter.

6. Implementation of rule-based modeling languages

Based on the formal semantics of ML-Rules, we now turn to ways to implement the language. First, we review existing simulation modeling paradigms as well as some existing approaches for simulation modeling with DSLs. Next, we sketch some basics of implementing ML-Rules, in particular the pattern matching. This is necessary as the formal semantics only specifies that all matches of a pattern in a current solution must be considered, but not how they are found. Therefore, we design a pattern matching algorithm for ML-Rules' reactant patterns. Using this algorithm, we then present two DSL implementations of ML-Rules. The first implementation is an internal DSL in Scala that employs the paradigm of functional programming. The second implementation is based on object-oriented principles and relies on the language workbench Xtext. We then evaluate both implementations by relating them to each other and to other DSL implementations of modeling languages. The chapter closes with an excursus that transfers the ideas developed for CTMC-based population models to rule-based agent-based modeling.

6.1. Simulation modeling paradigms and languages

So far we have discussed the mathematical and formal reasoning behind the stochastic simulation of a CTMC and how a CTMC is defined by a rule-based model. Now, we relate this approach to the general field of simulation methodology. This will illuminate the range of possible implementations of simulation approaches, and allow us to describe the implementation options for ML-Rules with more context.

We restrict our discussion to discrete-event simulation and omit continuous and discrete-stepwise simulation with fixed time steps. We justify this in two ways. First, languages such as ML-Rules are primarily executed by discrete-event simulation algorithms such as the SSA family. We have already seen that a continuous approximation of the SSA is possible (Section 3.2.3), and ODE- or PDE-based models find many uses for biological applications. Nevertheless, we consider the stochastic discrete-event simulation the ground truth and focus on it. Second, as argued by Law [133, pp. 78f], discrete-event simulation is a generalization of discrete-stepwise simulation, as fixed time steps can be emulated with artificial pseudo-events in a discrete-event simulation. In addition, fixed time steps lead to problems with ordering of events that occur at the same time, which happens more often with bigger time steps. Conversely, the computational efficiency suffers increasingly from smaller time steps [133, pp. 78f].

6.1.1. World views

There are three different "world views" on discrete-event simulation: event scheduling, process interaction, and activity scanning [14, p. 114, 178, p. 22, 156].

In the event scheduling approach, events and their timing are in the focus. A model is specified in terms of causal and temporal dependencies between events. For example, in a queuing model the arrival of a customer can be modeled as an event that schedules a future event for the arrival of the next customer after a stochastic delay sampled from some distribution. Similarly, depending on whether a server is currently free, an event for finishing service for the customer can be scheduled and the server is marked as occupied; otherwise, a state variable that holds the number of waiting customers is updated. These dependencies between events and the current model state can be expressed with formalisms like event graphs. In the context of simulating biochemical reaction networks, event scheduling is most closely related to the next reaction method (Section 3.2.2). However, here the actual algorithm is an additional layer of abstraction between actual events and the model (which is not concerned with events at all).

The process interaction approach focuses on the key entities in the model and their life cycles. A process is then the development of an entity during the simulation, consisting of states, events, or demands for shared resources. This can, for example, be expressed in a flow chart, where blocks representing the life cycle stages are connected with directed edges [14, p. 140]. Taking again a queuing model as an example, a customer could be modeled as an entity whose life cycle consists of stages such as arrival, being served by a server (for some period of time), and leaving the system. The actual queuing and competing for the servers would be handled by the simulation engine. Thus, the process interaction world view allows for a higher-level, more abstract model description than the event scheduling approach. For simulating biochemical reaction networks some approaches that take the process interaction world view have been proposed. The most immediate implementation of this world view can be found in process algebras, for example the attributed π-calculus [114]. The DEVS framework introduced by Zeigler also uses the process interaction approach and allows for modeling biological entities (such as cells) as hierarchical, reactive systems [212, 100, 77].

Finally, in the activity scanning world view [164, 165] a model is specified by a number of activities that are constrained by preconditions. To find the activities that occur at a specific simulation time, the preconditions are evaluated on the model state at that time. The original description of activity scanning used a fixed time increment, but it was later combined with event scheduling to form the so-called "three-phase" approach. Here, the preconditions of the activities are only checked at times at which the model state changes due to unconditional, pre-scheduled events. Petri nets are a typical example of modeling approach based on activity scanning and have been applied for modeling of biochemical reaction networks [141].

All three world views have in common that they have a declarative core and even map to graphical formalisms such as event graphs, flow charts, or Petri nets. The fundamental difference is *where* the model's behavior is specified [212]. It can be part of the description of the entities as in the process interaction world view, or it can be described as global events that affect the entities as in the event scheduling and activity scanning world view.

The rule-based nature of ML-Rules and similar languages is best reflected in the activity scanning world view. Here, reaction rules are defined globally and affect all entities in the model, including entities in nested compartments. However, ML-Rules has also been used to develop individual-based models [190], which is more closely related to the process interaction world view.

This duality also allows approaching calculation of reaction propensities as discussed in Section 3.3.2 from another angle. In a global top-down perspective, such as the activity scanning world view, instantiating a reaction can be thought of as simultaneously selecting the reactants from the available entities. This leads to counting the distinct unordered reactant sets (the number of combinations in combinatorics). On the other hand, in an individual-based perspective like the process interaction world view, the mental picture might be that the first reactant initiates the reaction by selecting a second entity, then both selecting the third and so on. In that case, the first reactant is different from the second one, and the order of reactants matters. This leads to counting permutations of reactants via the falling factorial.

ML-Rules does not attempt to resolve this ambiguity. Instead, it provides the means to explicitly specify how the rate is to be computed and, thus, which perspective or world view the model is considered from (and whether to count combinations or permutations). This expressive power is part of ML-Rules' DSL nature. Next we look into concepts for implementing simulation modeling languages as DSLs.

6.1.2. Simulation modeling with DSLs

Relating simulation modeling paradigms and programming language paradigms allows us to reason about the pragmatic implementation of simulation applications [79]. For example, implementing an agent-based model might benefit from using an object-oriented programming language, as it already contain some of the necessary features: attributed entities, encapsulation, or message passing. Conversely, models where pipelines of transformations are in the focus can be implemented using the function composition of functional programming languages.

One conclusion from this insight might be to extend existing programming languages with simulation-related constructs, that is, construct internal DSLs for implementing models. One example for this approach in Python(P)DEVS [227]. Here the abstract, mathematical syntax of DEVS is mapped to a concrete syntax in Python. For example, an atomic DEVS model is represented by a class with internal state and methods to implement DEVS elements like the internal transition δ_{int} or the time advance ta. Superclasses for atomic and coupled models encapsulate reusable code and establish an interface for the provided simulation algorithms. This way, Python(P)DEVS employs pure object-oriented programming to specify the relation between the language implementation and the model as well as between the model components.

Another example is ScalaTion [155]. ScalaTion is a Scala-based internal DSL that supports writing "simulation programs". This way, ScalaTion attempts to combine the domain-specificity of tailored Simulation Programming Languages with the performance, expressive power, and library ecosystem of GPLs. However, in contrast to the declarative formalisms that take the above world views, ScalaTion still requires writing the model as a program rather than a declarative specification. The model is written in a combination of functional and object-oriented elements, applying them as appropriate for the individual world views. Consequently, ScalaTion relies on standard internal DSL design and implementation techniques, for example class inheritance, operator overloading, and closures.

ScalaTion and Simulation Programming Languages are tailored to the domain "discrete-event simulation". Python(P)DEVS is more specific and focuses on one formalism for discrete-event simulation. Even more specific are modeling languages that focus on a more specific application domain to more precisely capture the idiosyncrasies of that domain[1]. Such languages often aim for a declarative model specification and, thus, are implemented as external rather than internal DSLs. This allows adopting declarative syntax that would be hard to integrate in a GPL. For example, in the QUANTICOL project, the implementations of the "Process Algebra of Located Markovian Agents" (Paloma) and the language CARMA ("Collective Adaptive Resource-sharing Markovian Agents") are external DSLs based on Xtext [76, 31]. Another example for an Xtext-based simulation modeling language is the Complex Adaptive Systems language (CASL) [22]. Paloma is a syntactically very simple language that implements a process algebra approach and exploits the opportunity to define a standalone syntax for a entirely declarative language design. CARMA and CASL, on the other hand, follow an object-oriented language design, which is a common use case for Xtext and well supported. For example, constructing imperative sequences of statements or scoping rules for nested blocks are part of Xtext's standard documentation. Another example for an external DSL for modeling is Modelica, which is object-oriented and declarative, with declarativity mostly referring to the absence of assignment statements in favor of equations [83].

ML-Rules and the other rule-based languages reviewed in Section 4.2 are declarative with the reaction rule concept as the fundamental element. As described by Oury and Plotkin, this is closely related to stochastic term rewriting [176]. Term rewriting is well researched and some ideas and techniques can be transferred to rule-based modeling languages. For example, matching a rule to a solution in the current state of the model can be solved by unifying the pattern on the left rule side with the solution [10, p. 78].

As a next step, we give an overview of the general design of our implementation of ML-Rules. The rest of this chapter will explore how ML-Rules can be implemented as an internal DSL in the functional programming paradigm and as an external DSL on top of object-oriented concepts. As an excursus, we also investigate how the idea of rule-based modeling can be translated to an internal DSL for an object-oriented agent-based simulation framework, which adopts some ideas from the process interaction world view.

[1]To make this difference more obvious, we could call ScalaTion a "simulation language" in contrast to "modeling languages" that focus on a particular modeling domain.

6.2. Implementing ML-Rules

In general, the implementation of ML-Rules has two key players: the model and the simulator. The clear interface between both is a prerequisite for reusing simulation algorithms or executing models with different algorithms. The model is written in a DSL and contains at least the specification of the initial solution and the reaction rules that govern the model's behavior. Typically, a model also contains declarations of the species (with their typed attributes) as well as definitions of constants and functions. To execute a model, the simulator creates the initial solution as defined by the model and then repeatedly applies the reaction rules to the solution to obtain all possible reactions [221]. Then, as described in Section 3.1.4, the First Reaction Method or the Direct Method is used to probabilistically select a reaction to execute as well as a time advance[2]. This is repeated until some stopping criterion (for example the maximum simulation time) is reached. Algorithm 1 shows this procedure in pseudocode.

```
Input: An initial solution $sol_{init}$, a set of rules $R$, and a stop time $t_{stop}$
1  $sol \leftarrow sol_{init}$
2  $t \leftarrow 0$
3  while $t < t_{stop}$ do
4      $reacs \leftarrow \emptyset$
5      foreach $r \in R$ do
6          foreach $s \in subsols(sol)$ do
7              foreach $reac \in match(rule, s)$ do
8                  $reac' \leftarrow lift(reac, s)$
9                  $reacs \leftarrow reacs \cup \{reac'\}$
10             end
11         end
12     end
13     $(\Delta_t, reac) \leftarrow select(reacs)$
14     $t \leftarrow t + \Delta_t$
15     $sol \leftarrow apply(sol, reac)$
16 end
```

Algorithm 1: The SSA as applied in ML-Rules

The algorithm references five further functions.

subsols obtains all solutions in the current model state. This includes the root solution as well all as recursively nested solutions.

lift modifies a reaction's rate and effect to lift it from a subsolution to the root solution as described in Section 5.4.

[2]For now, we do not consider other SSA variants such as the Next Reaction Method, where the obtained reactions and firing times are maintained between simulation steps (see Section 3.2.2). We discuss the potential for incorporating such modifications later on, in particular in Section 6.6.

match matches a rule to a solution, returning a set of reactions. Each reaction consists of a rate (a real) and a description of the changes to apply when the reaction fires. As we will see, these changes can be expressed in different ways, most directly as a change vector. Alternatively, the changes caused by a reaction can also be captured in a closure that mutates the state when executed.

select samples a reaction and a time advance according to the rates of the given reactions, for example with the First Reaction Method or the Direct Method.

apply executes the reaction and returns a modified solution. This can happen by mutating the solution in-place or by creating a new immutable solution value with the changes incorporated. In any case, the reaction must be applied to the (sub)solution in which it was matched.

Whereas most of this algorithm is standard, the functions *subsols* and *lift* are specific to ML-Rules. They take care of applying rules in every subsolution as defined by ML-Rules' formal semantics.

We did not include the output of the simulation in the pseudocode algorithm. Approaches to observe a simulation run differ in when to observe (at each state transition, at fixed time points, repeatedly after an interval) as well as in what to observe (the whole state, specific observables, complex queries including aggregation and filtering) [102]. However, this issue is orthogonal to the ones considered in this thesis and we will largely ignore it.

What is left to discuss independently of specific implementation choices is the realization of the function *match* above. In particular, we will specify an algorithm to obtain all variable substitutions that identify the left side of a given rule with a subset of a given solution as required by the formal semantics.

6.2.1. Pattern Matching algorithm

To match a rule against a solution, the variables in the rule need to be mapped to values in such a way that the substitution maps the pattern on the left rule side to the solution. This can be expressed as an equational unification problem, where the equations represent associativity and commutativity (AC) laws about the + operation on solutions and patterns[3] [10, pp. 223ff, pp.236ff]. See Section 5.5.4 for a short review of previous work in this area by Oury and Plotkin.

An equational unification problem in which terms only consist of variables, operators as specified by the equations, and additional constants can be solved by translating it to a system of diophantine homogeneous linear equations. There exists an extensive body of research on solving equational unification problems this way [10, pp. 262f]. However, as noted by Oury and Plotkin [176, 175], representing attributed entities and compartments additionally requires unary function symbols and colors. In addition, instead of unifying two terms which may each contain variables, we always only need to unify a ground term (the solution contains no variables) with a term that may contain variables. This problem

[3] Recall that patterns and solutions are mapped to multisets (Section 5.4).

is called equational matching and can, for example, be solved by compiling the pattern to a discrimination net [10, p. 229, 11, 128, 127].

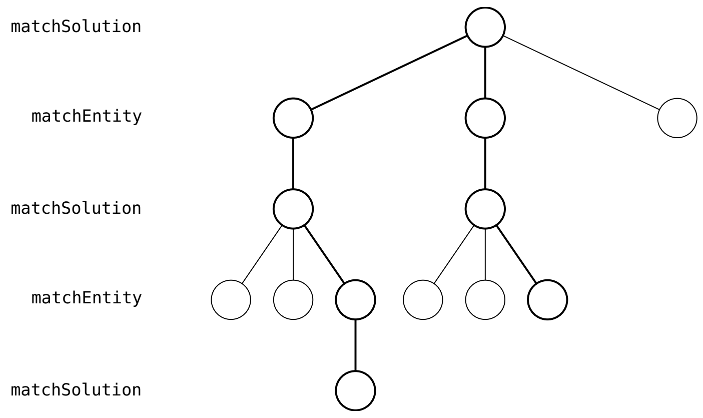
Formally, we can describe our pattern matching problem based on the definition of the abstract syntax in Section 5.4: Given a pattern pat_r and a solution $\mathcal{A}$ we are looking for all substitutions σ that satisfy $pat_r \Downarrow_{\mathcal{A},\sigma} \mathcal{A}$. It is also important to note that the patterns in ML-Rules are non-linear, meaning variables can occur multiple times on the left rule side. It has been shown that AC-matching for non-linear patterns is NP-complete, whereas linear patterns can be matched in polynomial time [18].

In this thesis, we use a simple recursive backtracking algorithm that traverses a pattern from left to right in preorder fashion. Before defining the algorithm formally, we give a short informal explanation. Essentially, applying the algorithm to a pattern and a solution can be visualized as a pattern matching tree (Figure 6.1). Each tree node corresponds to an intermediate result obtained by matching a prefix of the pattern. In the root node, the whole pattern is left to match the whole solution and no substitution for any variable has been determined. A successful match is a leaf node in which the whole pattern has been consumed and a substitution for all variables in the pattern has been determined. Unsuccessful branches do not lead to a leaf with an empty pattern. Thus, each node contains the remaining parts of the pattern to match, the solution of unmatched entities, and the substitution obtained so far including counts # for reactant variables.

The relation between a parent and a child node corresponds to consuming one pattern component and finding consistent substitutions for the variables contained in the pattern component. Existing substitution for variables that already occurred earlier in the pattern are obeyed[4]. For example, when matching an entity pattern to an entity, all attribute variables in the entity pattern are matched to the corresponding attribute values of the entity. This results in a substitution for all previously unused variables in the entity pattern. The tree branches when an entity pattern can be matched to several entities in the solution. Figure 6.1b shows one successful branch of matching a simple non-linear pattern. On the other hand, Figure 6.1c shows an unsuccessful branch, in which $B(x)$ can not be matched to $B(1)$ because x is already bound to 2.

The formal definition of the algorithm is structured into two mutually recursive functions `matchSolution` (Algorithm 2) and `matchEntity` (Algorithm 3). Initially, `matchSolution` is called with the complete pattern (i.e., the left rule side), the solution in which to find rule instantiations, and an empty substitution. `matchSolution` then calls `matchEntity` for each entity pattern and each entity in the solution (line 11). `matchEntity` in turn calls `matchSolution` in line 14 to match the sub-solution pattern of an entity pattern. In the process, the substitution is extended with newly matched variables, including variables used for entity attributes, for entities, and for subsolutions, as well as the count values for entity variables of the form $\#x$. In addition, matched entities are removed from the solution making them unavailable for subsequent matches. Earlier variable bindings are obeyed. For example, in algorithm 2 line 4 checks whether the rest solution variable has already been bound to a value and, if so, the current rest solution is equal to that value.

[4]This aspect is specific to non-linear pattern matching, in which variables are allowed to occur several times.



(a) Schematic match tree for the example. The bold branches are written out in detail in Figure 6.1b (left branch) and figure 6.1c (right branch).

`matchSolution`$(3A(1) + 5A(2) + 2B(1), A(x) \triangleright a + B(x) \triangleright b + s, \{\})$

  `matchEntity`$(3A(1), A(x) \triangleright a, \{\})$

`matchSolution`$(2A(1) + 5A(2) + 2B(1), B(x) \triangleright b + s, \{x \mapsto 1, a \mapsto A(1), \#a \mapsto 3\})$

  `matchEntity`$(2B(1), B(x) \triangleright b, \{x \mapsto 1, a \mapsto A(1), \#a \mapsto 3\})$

`matchSolution`$(2A(1) + 5A(2) + B(1), s, \{x \mapsto 1, a \mapsto A(1), \#a \mapsto 3, b \mapsto B(1), \#b \mapsto 2\})$

$$\{x \mapsto 1, a \mapsto A(1), \#a \mapsto 3, b \mapsto B(1), \#b \mapsto 2, s \mapsto 2A(1) + 5A(2) + B(1)\}$$

(b) An example for a successful attempt of matching a pattern to a solution.

`matchSolution`$(3A(1) + 5A(2) + 2B(1), A(x) \triangleright a + B(x) \triangleright b + s, \{\})$

  `matchEntity`$(5A(2), A(x) \triangleright a, \{\})$

`matchSolution`$(3A(1) + 4A(2) + 2B(1), B(x) \triangleright b + s, \{x \mapsto 2, a \mapsto A(2), \#a \mapsto 5\})$

  `matchEntity`$(2B(1), B(x) \triangleright b, \{x \mapsto 2, a \mapsto A(2), \#a \mapsto 5\})$

$$\lightning$$

(c) An example for an unsuccessful attempt of matching a pattern to a solution.

Figure 6.1.: Matching the pattern $A(x) \triangleright a + B(x) \triangleright b + s$ to the solution $3A(1) + 5A(2) + 2B(1)$.

If that is not the case, the matching fails and an empty set is returned.

Input: A solution sol, a pattern with rest p_r, a substitution σ
Output: A set of substitutions

```
function matchSolution($sol, p_r, \sigma$):
    $p_1 + \cdots + p_n + x \leftarrow p_r$
    if $n = 0$ then
        if $x \in \mathcal{D}om(\sigma) \wedge \sigma(x) \neq sol$ then
            return $\emptyset$
        else
            return $\sigma \cup \{x \mapsto sol\}$
    else
        $result \leftarrow \emptyset$
        foreach $e \in sol$ do
            $matches \leftarrow$ matchEntity $(e, p_1, \sigma)$
            foreach $\sigma' \in matches$ do
                $sol' \leftarrow sol \ominus p_1\sigma'$
                $result \leftarrow result \cup$ matchSolution$(sol', p_2 + \cdots + p_n + x, \sigma')$
            end
        end
        return $result$
```

Algorithm 2: The non-linear pattern matching algorithm for solutions.

The algorithm is efficient in that it prunes subtrees that can not lead to a match anymore due to different values for the same variable. This is different from many existing algorithms, which first do linear pattern matching and then filter out all results with conflicting values for a variable [11]. An upper bound of the algorithm's run time is $O(m^n)$, where m is the number of entity patterns in the pattern and n is the number of entity populations in the solution and its subsolutions. Note that m is typically small, whereas n can become very large.

A final point regarding non-linear pattern matching is its generality. Non-linear patterns allow expressing different types of (spatial) relations between entities. As discussed when relating ML-Rules and $React(C)$ in Section 5.5.4, nesting can be taken into account via shared variables. For instance, the non-linear pattern $A(p) + A(p)$ (where the attribute of A represents the parent compartment) only matches pairs of As in the same compartment. Similarly, a non-linear pattern like $C(vol, p) + A(p) + A(p)$ makes the parent C entity and its volume attributes available for calculating a volume-dependent propensity. Through generating new variable values on the right rule side, dynamic nesting can be expressed as well (see Section 5.5.4). In addition, when allowing expressions in the patterns other spatial relations such as being neighbors in a grid can be expressed with non-linear patterns such as $A(x, y) + A(x + 1, y)$. This makes non-linear patterns an interesting avenue for future research (see Section 8.3).

A complete Scala implementation of the algorithm presented in this section is given in

```
Input: An entity $e$, an entity pattern $p$, a substitution $\sigma$
Output: A set of substitutions
1  function matchEntity($e, p, \sigma$):
2      $c_e S_e(a_{e,1}, \dots, a_{e,n}; sub_e) \leftarrow e$
3      $c_p S_p(a_{p,1}, \dots, a_{p,n}; sub_p) \triangleright x \leftarrow p$
4      if $S_e \neq S_p \lor c_p > c_e$ then
5          return $\emptyset$
6      else if $x \in \mathcal{D}om(\sigma) \land \sigma(x) \neq c_p S_e(a_{e,1}, \dots, a_{e,n}; sub_e)$ then
7          return $\emptyset$
8      else
9          $result \leftarrow \emptyset$
10         $\sigma' \leftarrow \sigma$
11         foreach $i \in \{0, \dots, n\}$ do
12             $\sigma' \leftarrow$ matchAttribute($a_{e,i}, a_{p,i}, \sigma'$)
13         end
14         $sub \leftarrow$ matchSolution($sub_e, sub_p, \sigma'$)
15         foreach $\sigma'' \in sub$ do
16             $result \leftarrow result \cup \{\sigma'' \cup \{x \mapsto c_p S_e(a_{e,1}, \dots, a_{e,n}; sub_e), \#x \mapsto c_e\}\}$
17         end
18         return $result$
```

Algorithm 3: The non-linear pattern matching algorithm for entities.

Appendix B.1.

6.2.2. Differences between formal semantics and implementation

In Algorithm 3 line 12 refers to a function `matchAttribute`, which is not shown. This function deterministically matches an attribute pattern to an attribute value. The abstract syntax allows arbitrary expressions for each attribute of an entity pattern. For example, given a species A with one `real` attribute and a function f : `real` $\rightarrow$ `real`, the following left rule side would be valid according to the abstract syntax:

$$A(f(x); s_1) + s \rightarrow A(x; s_1) + s$$

The formal semantics now states that for all values of x for which $A(f(x); s_1)$ matches an entity in the current solution, a reaction gets instantiated. Without analyzing (and possibly inverting) f, a pattern matching algorithm can only find the right values for x by trying out all possible values in the domain of f. In this case though, this is the type `real`, which has an infinite number of values! For this reason, the pattern matching in all our ML-Rules implementations allows only constants or variables for attributes in species patterns. Thus, the concrete syntax is smaller than the abstract syntax to avoid patterns that can not be matched with reasonable computational effort. We allow this difference between concrete and abstract syntax, as it does not impose limits on the abstract syntax and also allows to specify it succinctly. Future implementation might come closer to including more of the abstract syntax[5]. In any case, the abstract syntax gives us a way to precisely say which aspects are implemented and which are not.

Another noteworthy aspect is that # now maps each entity variable to the number of available entities *at the time of matching*. This also means a further difference from the formal semantics. As discussed in Section 3.3.1, this facilitates correctly integrating mass action kinetics when calculating reaction rates. It is also possible to translate back and forth between both styles of determining the population sizes [106]. The numerical difference is the count of the corresponding entity patterns. The cases where the calculation is more complicated are when a rule can yield homoreactions and heteroreactions, for example with a left side like $A(x) + A(y)$.

Consider the rule

$$A(x) \triangleright a_x + A(y) \triangleright a_y \xrightarrow{r \cdot f(x,y,\#a_x,\#a_y)} A(x+y),$$

where f shall calculate the correct rate factor according to mass action kinetics (see Sections 3.3.1 and 5.3). With the definition # as in the formal semantics, the definition of f would need to be

$$f(x, y, n_x, n_y) = \begin{cases} n_x \cdot n_y & x \neq y \\ n_x \cdot (n_x - 1) & x = y. \end{cases}$$

With the # function as yielded by the pattern matching algorithm, which assigns to $\#a_y$ the number of $A(y)$ entities that remain after removing the first reactant $A(x)$, it is just

$$f(x, y, n_x, n_y) = n_x \cdot n_y.$$

No distinction of cases is necessary. Therefore, we argue that the version of the # function as yielded by the pattern matching algorithm is more practical than the one in the formal semantics. Again, allowing this difference between the formal semantics and the practical implementation enables succinctness in both worlds. At the same time, we can precisely define the difference and, if necessary, translate between both worlds.

Implementing pattern matching is a big part of any implementation ML-Rules (or other sufficiently expressive rule-based languages). The precise definition of abstract syntax and formal semantics allows us to precisely describe how a concrete implementation relates to the formal semantics. We saw a few examples of differences between the presented pattern matching algorithm and the formal semantics. The reason is that the formal definition of the language and the implementation have different goals. Whereas the formal definition gives an unambiguous description of the capabilities of the language, the implementation aims at being usable in practical applications. To be succinct in both worlds, slightly different approaches can be taken.

Allowing differences between formal definition and implementation does not mean that one of those is wrong. On the contrary, the formal definition gives us a vocabulary to

[5] Chromar [106], for example, allows patterns such as $A(x) + A(f(x))$. The pattern is transformed to $A(x) + A(y)$ with a constraint $y = f(x)$, which is evaluated after values for x and y have been found. As noted below, this is a usual approach for translating a non-linear pattern to a linear one [11].

precisely describe how the implementation differs. In some cases we can even define a bidirectional translation between a concrete implementation and the formal semantics. For example, for the count function # a case distinction as shown above can handle this translation. Of course, at some point the difference can get too big to still claim that an implementation corresponds to a formal semantics. In our case, however, the core aspects of ML-Rules are maintained: reaction rules on attributed, nested entities with custom rate functions. Nevertheless, some of the properties that can be proved based on the formal semantics may not hold anymore in the practical implementation, for example the irrelevance of reactant order.

By not requiring a complete match between formal definition and implementation we also allow for several implementations with different coverage of the semantics. This can be useful in two ways. First, this allows providing specialized algorithms tailored to models that do not exploit all language features [101]. Second, it facilitates the evolution of simulation algorithms, gradually increasing the coverage of the semantics. In both cases, implementations can be assessed and compared precisely using the formal semantics. Performance-wise, this means that we can precisely say which aspect of a modeling language makes our simulator slow (e.g, non-linear pattern matching, unbounded domains for attributes). This enables assessing the trade-off between expressive power and performance (cf. Section 2.5.3).

6.2.3. Network-based simulation

The simulation algorithm as presented in algorithm 1 is not the only way to execute models defined in ML-Rules or similar languages. In this section, we give a short overview over an alternative way to execute rule-based models.

Our approach is a so-called *network-free* simulation algorithm [218]. Here, the reactions that are possible in the current state are generated "on-the-fly" by pattern matching the reaction rules against the current state. In contrast, the so-called *network-based* simulation algorithms use the rules only once to construct a reaction network before running the actual simulation, which is then purely based on the static network [94]. This essentially reduces the rule-based model to a reaction-based one. However, reaction networks can be too large (or even infinite) to be exhaustively generated (for example, because of attributed species, cf. Section 4.1.2).

Often, it makes sense to provide both network-based and network-free simulation algorithms for a language. For example, the original BioNetGen[6] simulator for BNGL was network-based, and only later the network-free simulator NFSim was developed [218]. The generation of the network also relies on applying the rules to a solution. Starting with some seed species, the generation of all reactions is repeated until a fixpoint or some termination criterion is reached [73]. Thus, the pattern matching algorithm as shown in the previous section are also needed for network-based simulators.

In the remainder of this chapter, we will focus on network-free simulation. To illustrate how different implementations of ML-Rules and, in particular, the proposed pattern

[6]The (originally) network-based nature of BioNetGen is evident from its full name "Biological Network Generator" [71, p. 117]

matching algorithm are possible, we now present two variants that follow opposite approaches in many ways. First, a purely functional, Scala-based internal DSL, and second an external DSL that relies on generating object-oriented Java code.

6.3. ML-Rules as an internal DSL in Scala

Scala is a GPL that allows programming in the functional and in the object-oriented paradigm. As a consequence, it offers many programming concepts that can be used to implement internal DSLs. In particular, Scala provides powerful pattern matching capabilities as typical for functional programming languages. The main idea of the approach described in this section is to implement ML-Rules as an internal DSL in Scala, which allows reusing Scala's built-in pattern matching. Based on a short introduction to the available pattern matching features in Scala we focus on the implementation of the pattern matching algorithm as described in Section 6.2.1.

6.3.1. Pattern Matching in Scala

Scala offers a succinct syntax for matching patterns with variables to values [171, p. 273]. For example, the following code snippets matches three variables in a tuple pattern to according values from a given tuple:

```scala
val (x, y, z) = (1, "a", true) // x == 1, y == "a", z == true
```

The same is possible for algebraic data types, which are implemented in Scala as *case classes* for product types and *sealed traits* and subtyping for sum types [170]. `match-case` expressions can be used to distinguish between sum types. In the following snippet, the sealed trait `Animal` is a sum type with the subtypes `Cat` and `Dog`, both of which have an attribute `name`.

```scala
sealed trait Animal

case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal

def speak(a: Animal) = a match {
  case Cat(n) => n + " says Meow"
  case Dog(n) => n + " says Woof"
}
```

Patterns can also include the wildcard pattern _, types, or variable bindings, include guards, or be nested. This makes pattern matching very powerful in functional languages such as Scala, particularly in combination with algebraic data types. In addition, case classes are automatically augmented with generated code by the Scala compiler, for example for constructing and comparing instances. All attributes of a case class are immutable by default.

We can leverage Scala's powerful pattern matching to help implementing the pattern matching required for ML-Rules. This allows for a very succinct implementation that relies on established functional programming patterns. We first describe an implementation without nesting, and then extend it to include nesting.

6.3.2. Unnested multisets

As a first step, we define some types that will represent solutions and entities, similarly as in the formal semantics. Solutions will be represented as multisets. A multiset of `A`s is a map of `A`s to the count of occurrences. Operations such as adding and subtracting multisets can be easily implemented based on this type alias.

```
type MultiSet[A] = Map[A, Int]
trait Entity
```

We also define a trait `Entity`, which is the supertype for the user-defined case classes that represent biochemical entities. In a model with entity types `A` and `B`, where `A` has one attribute and `B` has two, a solution could look like this:

```
case class A(x: Int) extends Entity

case class B(x: Int, y: Boolean) extends Entity

val solution: MultiSet[Entity] = Map(
  A(1) -> 2,
  A(2) -> 2,
  B(0, true) -> 1
)
```

This translation from the abstract syntax to Scala is pretty straightforward, and we will later improve it further by adding syntactic sugar for constructing multisets. Note, however, that Scala requires naming attributes in contrast to ML-Rules. As we do not use the attribute names in the following, they can be chosen at will (cf. 4.1.2).

The final ingredient for defining a first approximation of a reaction rule are for-comprehensions. Among other things, Scala's for-comprehensions allow iterating over collections. By applying this to the entities in a multiset and combining it with the pattern matching on case classes, we obtain a first attempt of expressing a reaction rule:

```
for {
  A(x) <- solution.keys
  A(y) <- solution.keys
} yield A(x + y)
```

Each line between the braces `{}` corresponds to one reactant, and the variables `x` and `y` are vanilla Scala variables. Left of the arrow `<-` a Scala pattern makes sure that we only match `A` entities; the `B` entities in the solution are ignored. For each successful match of both reactants, the `yield` expression is evaluated and the evaluation results are accumulated. In the `yield` expression the variables `x` and `y` matched in the reactants are available. Thus, some aspects of the pattern matching algorithm from Section 6.2.1 are already available at this stage (without any non-trivial implementation). However, some other aspects are missing and will be addressed in the remainder of this section, before we go on to integrate nested entities.

- In addition to the variables for attributes of entities, we also want to obtain variables for the reactants themselves and the number of their available occurrences (for the # operator). Currently, the number of occurrences is even ignored completely.
- Currently, several reactants can match the same entity. We will make entities that have already been matched by a reactant unavailable for later reactants.
- In addition to the pattern matching on the left rule side, the rate expression and the right rule side need to represented.
- Scala's built-in pattern matching does not support non-linear patterns. We will discuss how ML-Rules' non-linear patterns can be expressed in our Scala implementation.

Reactant variables

Scala's built-in pattern matching supports assigning a variable to the value matched with `@`. For example, we can modify the snippet above to bind the variables `ax` and `ay` to the reactants:

```
for {
  ax@A(x) <- /* omitted */
  ay@A(y) <- /* omitted */
} yield /* omitted */
```

However, `ax` and `ay` are of the type `A` and it is not possible to augment them with additional information, in particular the count of available entities at the time of matching. To allow storing such additional information about a matched reactant we introduce the `Reactant` class. In addition, we add an *extractor object* [171, ch. 26] `|>` to allow an infix operator corresponding to the binding operator $\triangleright$ in the abstract syntax.

```
case class Reactant(entity: Entity,
                    count:  Int)

object |> {
  def unapply(reactant: Reactant): Option[(_ <: Entity, Reactant)] =
    Some((reactant.entity, reactant))
}
```

This way, we can augment Scala's pattern matching to allow binding a variable of the type `Reactant` with the following syntax:

```
for {
  A(x) |> ax <- /* omitted */
  A(y) |> ay <- /* omitted */
} yield /* omitted */
```

Now `ax` and `ay` are of the type `Reactant`. The expression `ax.count` then evaluates to the count of available entities at the time of matching, and, thus, corresponds to the abstract syntax expression $\#ax$[7]. The creation of the `Reactant` objects happens to the right of the binding arrow `<-`, where the code we discuss next is evaluated.

Avoiding repeatedly matching an entity

To avoid matching the same entity several times we need to maintain information about which entities have already been matched. More precisely, as we are in a multiset setting, we need to record how many occurrences of every entity have been matched. Then we can ignore the already matched entities in each subsequent matching step. The observation that pattern matching is a computation where earlier steps affect later steps is a strong hint that such a pattern matching computation is monadic. The idiomatic way to express monadic computations in Scala are for-comprehensions, which is similar to Haskell's do-notation.

Our for-comprehension above is essentially using the list monad, but, as we already noticed, this may result in using an entity for several reactants. Therefore, we apply the state monad transformer to the list monad. This way, we can express computations that are simultaneously stateful and nondeterministic. The resulting monad can be captured in a case class (Appendix B.2 shows a complete implementation)[8]:

```
case class NonDetState[S, +A](run: S => List[(S, A)]) {
  // omitted
}
```

Thus, `NonDetState` wraps a function that takes a state and returns a list of pairs, with each pair representing a possible successor state with a corresponding intermediate result. For matching entities in multisets, the state contains the available entities and also accumulates the entities that have already been matched. Omitting nested entities for now, both these components are just multisets of entities:

```
case class MatchingState(available: MultiSet[Entity],
                         taken:     MultiSet[Entity] = Map.empty)
```

Now we can express the core part of the pattern matching algorithm as a `NonDetState` value. For each population of entities that is available and not already completely taken,

[7] As shown below, this yields the count at the time of matching. Note, however, that it would be trivial to also make the total count available. This is an example for the extensibility of internal DSLs.

[8] Functional programming libraries such as cats (typelevel.org/cats/) offer more featureful implementations of monads and monad transformers. For example, these implementations typically employ trampolining for stack-safe recursion. Using such a library, the definition of `NonDetState` becomes simply `type NonDetState[S, +A] = StateT[List, S, A]`.

one pair of the state with the additional taken entity and the entity itself wrapped in a reactant is returned:

```
val solution: NonDetState[MatchingState, Reactant] = NonDetState { s =>
  for {
    (e, availableCount) <- s.available.toList
    takenCount          = s.taken.getOrElse(e, 0)
    count               = availableCount - takenCount
    if count > 0
  } yield (
    MatchingState(s.available, s.taken.updated(e, takenCount + 1)),
    new Reactant(e, count)
  )
}
```

This way, we can express matching several reactants for a rule with a monadic for-comprehension. Continuing our example from above, we can now fill in the gaps on the right of the arrows `<-` as follows:

```
for {
  A(x) |> ax <- solution
  A(y) |> ay <- solution
} yield /* omitted */
```

This implements the pattern matching on the left rule side. Each line in the for-comprehension corresponds to one level in the tree in Figure 6.1 on page 94. In particular, the `MatchingState` contains the information about the remaining solution, whereas the variable substitution is represented by the Scala variables and their values. Next we take a look at how the rate expression and the right rule side can be implemented to express a complete reaction rule.

Instantiating reactions

Rules not only define the reactants, but also the rate and the products of the resulting reaction. We represent a reaction as a case class with a rate and a change vector and define a `NonDetState` value `react` that produces a `Reaction`.

```
case class Reaction(rate:         Double,
                    changeVector: Map[Entity, Int])

def react(
  rate:     Double,
  products: MultiSet[Entity]
): NonDetState[MatchingState, Reaction] = NonDetState { state =>
  List((state, Reaction(rate, diff(products, state.taken))))
}
```

Then, we can express a reaction rule as a monadic sequence of matching reactants followed by instantiating the reaction. The rule in the following snippet corresponds to the abstract syntax $A(x) \triangleright ax + A(y) \triangleright ay \xrightarrow{k \cdot \#ax \cdot \#ay} A(x+y)$:

```
for {
  A(x) |> ax <- solution
  A(y) |> ay <- solution
  r <- react(rate = k * ax.count * ay.count, products = Map(A(x + y) -> 1))
} yield r
```

Note that in the arguments for `react` all variables from the pattern matching as well as the `.count` expressions for the reactant counts are available. The products of the reactions are specified as a map and, therefor, somewhat verbose. We will introduce some syntactic sugar for reaction products later.

Non-linear patterns

ML-Rules requires nonlinear pattern matching, that is pattern matching with multiple occurrences of the same variable (see Section 6.2.1). Scala and Haskell do not support nonlinear pattern matching[9], other languages such as Prolog or Erlang do. For example, the following snippet causes a compile error in Scala:

```
val (x, x) = (1, 1)
```

In general, it is an error in Scala to use the same pattern variable several times in one line in a for-comprehension. The most direct way to work around this limitation is to use different variables and add a guard that requires both variable values to be equal. Nevertheless, it is possible to use a pattern variable from an earlier line in a for-comprehension as a constraint. To distinguish a reference to an earlier variable from a newly defined variable, references need to be enclosed in back ticks [171, p. 281].

```
case class A(x: Int, y: Int) extends Entity

for {
  A(x, x) |> _ <- solution     // compile error, x is used twice
} yield /* ... */

for {
  A(x1, x2) |> _ <- solution
  if x1 == x2                  // this works
} yield /* ... */

for {
```

[9]One of the reasons is that to implement it, it must be possible to compare the attributes of algebraic data types, which is not always possible in these languages (for example, functions are not generally comparable). The topic has been discussed on the Haskell-cafe mailing list (`mail-archive.com/haskell-cafe@haskell.org/msg59617.html`).

```
  A(x, y) |> _ <- solution
  A(x, y) |> _ <- solution     // x and y are new variables that shadow the earlier ones
} yield /* ... */

for {
  A(x, y) |> _     <- solution
  A(`x`, `y`) |> _ <- solution // x and y refer to the already matched occurrences
} yield /* ... */
```

Discussion

This completes the implementation for flat, unnested multisets. From a DSL perspective, there is not much syntax or semantics on top of regular Scala. However, the for-comprehensions allow reading a rule as "for the entity . . . and the entity . . . the reaction . . . is possible". This emphasizes the stepwise selection of reactants in the pattern matching algorithm, rather than being declarative like the left rule side in the abstract syntax. Before extending this approach to allow for nesting, we discuss some properties of this style of rule formulation.

First, the nondeterministic state monad has been presented before, for example to solve constraint satisfaction problems [9]. Similarly, sampling without replacement can be realized this way, for example for drawing poker cards from a deck [134]. By adapting those approaches for working with multisets rather than lists, we arrive at the solution presented above.

Second, functions that operate on the monadic values defined above can be used to quickly add features. For example, the abstract syntax of ML-Rules includes reactants with a count greater than one. On way to implement matching n similar entities is is to repeat the matching step n times and check that all n matched entities are equal.

Third, Scala's for-comprehensions give very powerful syntactic support for rule formulations with very little implementation effort. All expressions are typechecked by the Scala compiler, IDEs provide editing support, and generally Scala features can be seamlessly integrated with rule definitions.

As a preliminary conclusion, we can state that Scala's pattern matching relieves us of a large part of implementing the matching of the left side of a rule against the current state. The built-in limitations can be worked around in a straightforward manner. Thus, the main work consists in translating the matching to the nondeterministic state monad, which also required only few lines of code. As a next step, we extend the approach to include nested multisets and implement the full semantics of ML-Rules.

6.3.3. Pattern matching in nested multisets

To extend the implementation to nested multisets, two aspects need to be revised. First, the representation of model states must account for nesting. Second, rules must be able to specify the nesting of reactants.

Using algebraic data types (and case classes in particular), we can model the nested multisets that make up an ML-Rules state as follows:

```
case class Solution(contents: Map[Compartment, Int])
case class Compartment(entity: Entity, subSolution: Solution)
case class Path(steps: List[Int])
```

Multisets are wrapped in a `Solution` type and now contain `Compartment` instances. One `Compartment` consists of an entity and a subsolution which is again a `Solution`. There is always one root solution, and all nested solutions are identified unambiguously by a path of `Compartments` starting at the root solution[10]. In particular, the root solution corresponds to the empty path.

This representation of states corresponds exactly to the specification of model states in the formal semantics. There we defined that the multiset elements have the general form $S(\tilde{v}; v)$, where $\tilde{v}$ are the attributes of the entity and v is the subsolution. The Scala equivalent is `Compartment(S(att1, att2, ..., attn), Solution(...))`, where `S` is a case class and a subtype of `Entity`.

Furthermore, we also need to adapt the `MatchingState`, which now, in addition to the root solution, maintains a map of the solution of already taken entities per path starting at the root solution. The starting point for defining a rule is still a `NonDetState` value solution that produces a `Reactant`. Reactants are now parameterized with the path to the (possibly nested) solution in which they were matched. To allow accessing the entities in a subsolution of a reactant as well as the rest solution in that subsolution, we extend the `Reactant` type with methods `subSol` and `rest`.

```
case class MatchingState(root: Solution, taken: Map[Path, Solution])
def solution: NonDetState[MatchingState, Reactant]
case class Reactant(compartment: Compartment, count: Int, location: Path) {
  def subSol: NonDetState[MatchingState, Reactant]
  def rest:   NonDetState[MatchingState, Solution]
}
```

The implementations of these functions are similar to the ones for the unnested case, but additionally need to handle the paths to the individual solutions. The method `subSol` allows iterating the subsolution of a compartmental entity. The method `rest` yields the rest solution in a compartment and makes it unavailable for future matches. Therefore, it should be used last in a sequence of matches in a solution, as also specified by the abstract syntax in Section 5.4.

As an illustration of how a reaction rule from a nested model can be expressed in this approach, the following listing shows a rule for the reaction number 2 from the Wnt model (see Section 4.1). The definition of the rate and of the reaction result has been factored out into individual lines. We also introduced some syntactic sugar to facilitate the creation of solution values as reaction products, for example with the infix operator `+` (cf. Section 2.5.2).

[10]Note that this assumes a stable enumeration of key-value pairs in the `contents` of a `Solution`.

```
for {
  Wnt()            |> w <- solution
  Cell(phase, vol) |> c <- solution
  Axin("p")        |> a <- c.subSol
                      s <- c.rest
  rate     = c.count * ((kApA_act * w.count * a.count) / vol)
  products = Wnt() + Cell(phase, vol).withSubSol(Axin("u") + s)
  r <- react(rate, products)
} yield r
```

6.3.4. Links

As a final ingredient, we show how links can be realized in this approach. First, we define a simple sum type for link values. A link is either free or linked with an identifying ID, which is randomly generated for new link values.

```
sealed trait Link

case object Free extends Link
case class Linked(id: UUID = UUID.randomUUID()) extends Link
```

Then we can define an entity type that has a link attribute and define rules that link free entities and break up linked entities. Note that only one new `Link` is created, saved in a `val`, and then used as an attribute value in both entities. This is idiomatic Scala for a let-in expression and implements the ν operator from the formal definition of ML-Rules. In the second rule, a non-linear pattern is used to match entities with the same link value.

```
case class B(x: Int, l: Link) extends Entity

val linkRule = for {
  B(x1, Free) |> b1 <- solution
  B(x2, Free) |> b2 <- solution
                 r  <- react(1, { val link = Linked(); B(x1, link) + B(x2, link) })
} yield r

val breakUpRule = for {
  B(x1, l@Linked(_)) |> b1 <- solution
  B(x2, `l`)         |> b2 <- solution
                        r  <- react(1, B(x1, Free) + B(x2, Free))
} yield r
```

6.3.5. Discussion

This first implementation of ML-Rules is based on expressing the pattern matching algorithm as described in Section 6.2.1 with pure functional programming. The non-deterministic state monad allows a direct formulation of several consecutive steps of matching reactants. By employing Scala's syntactic support for monadic programming through for-comprehensions as well as established libraries for functional programming, the implementation is relatively succinct. On the other hand, the concrete syntax for reaction rules is noisier than the abstract syntax of ML-Rules. In particular, the focus on stepwise matching of reactants leads to less declarativity.

Another important point is the limited computational efficiency. For example, matching several reactants at once (in other words, matching a reactant with a count > 1) is, in the monadic framework, most directly expressed by repeatedly matching a single reactant. However, this requires filtering the resulting lists of reactants for those where all reactants are actually equal. With more control over the matching process, this could be significantly simplified by selecting all required equal reactants at once. More generally, rule definitions can not be inspected for optimizations.

The underlying issue is that the data in each line of the form `pattern <- solution` only flows in the direction of the arrow, from right to left. Thus, the iteration of the available entities on the right side of the arrow has no access to the pattern. Therefore, the selection of potential reactants can not be tuned to the entities described by the pattern, for example a given reactant count. This is similar in the Haskell embedding of Chromar [106] (also see Section 4.2.5), which also reuses the pattern matching capabilities of its functional host language. The difference to Chromar is the semantics of the expression to the right of the arrow `<-`. Whereas Chromar uses the list monad and calculates reaction multiplicities as discussed in Section 3.3.2, our implementation relies on the nondeterministic state monad to keep track of which entities are no longer available for matching. This way, ML-Rules gives access to the counts of the reactants at the time of matching, allowing to express custom rate functions based on these counts (as discussed Section 6.2.1). In addition, Chromar provides an additional syntactic layer to mimic the rule syntax, whereas ML-Rules directly exposes the monadic sequencing of steps. Thus, ML-Rules is more verbose, but direct access to the monad allows factoring out recurring steps or applying library functions.

In summary, this implementation of ML-Rules shows the typical trade-off of internal DSLs. It is easy to reuse features of the host language, as we did for defining species as case classes, or by reusing the built-in arithmetic operators. Also, Scala's type system and established IDEs are available, and it is possible to integrate user-defined functions or third-party libraries. On the other hand, the potential for inspecting and analyzing models is limited. By reusing features of the host language and delegating specific language aspects to it, our implementation gives up the opportunity to influence the evaluation of these language aspects. For example, by expressing rates as plain Scala expressions it is not possible to infer how the rate depends on the reactant's population sizes, which would be necessary to apply a partial-propensity SSA [174]. As such, this implementation works, but offers little potential for more sophisticated execution algorithms.

To address these issues, we now present an implementation of ML-Rules as an external DSL. This allows fine-grained inspection of all syntactic elements and, therefore, does not constrain the choice of evaluation or execution strategies. It does, however, require more effort for the actual implementation.

6.4. ML-Rules as an external DSL

The second approach to implementing ML-Rules relies on the language workbench Xtext and the associated techniques as described in Bettini's textbook [21]. In particular, the pattern matching for a given reaction rule is implemented by generating according code. As arbitrary code can be generated, we can employ the pattern matching algorithm almost exactly as presented in Section 6.2.1. However, before we discuss the code generation, a few other aspects of the language implementation must be handled, most of which we got "for free" in our previous implementation. Whereas Xtext provides support for many of these aspects, ML-Rules' idiosyncrasies, in particular the non-linear patterns, require some adaptations of Xtext's defaults.

First, we create an annotated grammar, which is the foundation of an Xtext-based language. Xtext uses this grammar to generate a parser for the concrete syntax as well as a metamodel of Java classes that represent the abstract syntax. In the second step we manually adapt the generated metamodel to ML-Rules' non-linear patterns. Third, we discuss how a model is type-checked, which is again non-standard because of ML-Rules' pattern-based reaction rules. Based on a type-checked model, we then turn to generating code. The idea here is that a the code generated for a type-checked model should always be error-free. In addition to the generation of the actual pattern matching code, we also consider the interface between generated and hand-written code, such as simulation algorithms.

These implementations are integrated into the framework provided by Xtext, following the "hollywood principle" (Don't call us, we'll call you). As a result, Xtext can generate an Eclipse plug-in that seamlessly integrates the editing of ML-Rules models as well as the generation and compilation of code into Eclipse's build process. Consequently, the preferred target language for code generation is Java, the language that is commonly used with Eclipse.

6.4.1. Concrete Syntax

In contrast to the previous approach, the usage of Xtext allows us to freely design the concrete syntax without being limited by existing language syntax. On the other hand, the preexisting implementation ML-Rules 2 already defined a concrete syntax before this thesis [98]. For compatibility, we largely adopt the existing concrete syntax. Thus, a model file consists of constants, functions, and species, then an initial solution, and then the reaction rules.

One diversion from the previous approach is the incorporation of user-defined functions. In ML-Rules 2, all functions needed to be defined directly in the model in a Haskell-like syntax. In contrast, we allow such definitions only for simple expressions, for example for rate expressions. More complex functions that need to iterate solutions, obtain the species name of an entity, or access its attributes are now implemented in Java in a separate file against a generated interface. This is based on the observation that such functions on solutions were very hard to implement directly in ML-Rules, as little support (e.g., for debugging) is available.

As in the previous section, we focus mainly on the definition and implementation of rules and the matching of the left rule side to the model state. Our implementation with Xtext follows the approach presented in Section 2.5.1. The following is a simplified version of the Xtext (grammar) rule that specifies the concrete syntax of (reaction) rules:

```
Rule:
  (reactants+=SpeciesPattern ('+' reactants+=SpeciesPattern)*)?
  '->' products=Expression? '@' rate=Expression ';';
```

When Xtext processes this grammar rule, two things happen. First, in the metamodel a class[11] `Rule` representing rules is generated with three attributes `reactants`, `products`, and `rate` of the types `List<SpeciesPattern>`, `Expression`, and `Expression`. Second, the parsing machinery for rules is generated. Part of that machinery are links to the code that was generated for other grammar rules (`List<SpeciesPattern>` and `Expression`) as well as checking for syntax elements such as `->`, `@`, or `;`. Then, parsing a rule like

```
Sheep:s + Wolf:w -> Wolf + Wolf @ #s * #w * wolfGrowth;
```

results in the instantiation of an object of the type `Rule` with attributes that hold the information about the patterns, reactants, and rate expression of the rule as parsed according to the corresponding grammar rules. Overall, about 40 grammar rules define the concrete syntax of ML-Rules as well as the generated metamodel.

Most of the grammar rules are either adapted from standard programming language aspects (nested arithmetic expressions are the canonical example) or from the abstract syntax definition of ML-Rules. However, as Xtext is based on Antlr 3, an LL parser, left-recursion has to be avoided (see Section 2.2.1). This requires some translation from left-recursive abstract syntax rules to the grammar rules. For example, among the different alternatives for defining an expression is $eS(\tilde{e}, e)$, meaning an entity definition with a count expression in front of it. The direct translation of this rule is left-recursive and not accepted by Xtext:

```
Expression:
  /* other alternatives */ | count=Expression entity=Entity;
```

Instead, the grammar rule must be reformulated as follows:

```
Expression:
  (/* other alternatives */) ({EntityExpression.count=current} entity=Entity)?;
```

This allows all other kinds of expression either on their own or as the count of an entity definition. The assignment in curly brackets `{}` assigns the already parsed expression to the `count` attribute of an `EntityExpression` (a metamodel class) if and only if an entity is parsed after the expression. This way, an Xtext grammar can be expressed without left recursion and at the same time the resulting metamodel represents exactly the elements defined in ML-Rules' abstract syntax.

[11]Technically, an interface (`Rule`) and an according implementation (`RuleImpl`)

6.4.2. Metamodel customizations

Xtext generates a metamodel based on the grammar that also defines the concrete syntax. Therefore, the generated metamodel contains precisely the elements that are represented in the concrete syntax. For typical programming language constructs this is often the desirable behavior. For example, the metamodel generation supports declaring variables and referring to them:

```
VariableDef:
  name=ID;

VariableRef:
  var=[VariableDef];
```

Here, the grammar states that variable definition consist of an ID literal, whereas variable references link to a variable definition via brackets `[]`. As a consequence, when parsing a `VariableRef` Xtext expects to parse an ID literal that is the name of some `VariableDef`, as links with `[]` refer to the `name` attribute of the target by default. Thus, in the concrete syntax variable definitions as well as variable references consist of an ID literal. In the abstract syntax, however, a variable reference contains a link to a variable definition instead of an ID literal.

For ML-Rules, this automatic generation of references is useful, for example, for defining and referring to constants or functions, which works similarly as in GPLs. The DSL aspect of ML-Rules, however, shows in the patterns on the left side of the reaction rules. Here, there is not necessarily a clear distinction between defining a variable and referring to it. For example, consider the following rule with a non-linear pattern on its left side:

```
A(x):a + B(x):b -> C(x) @ #a * #b;
```

Whereas the variables `a` and `b` occur only once on the left side, the variable `x` occurs twice. There is no clear way to state which of the two `x` occurrences defines the variable. At first glance, it might seem sensible to make the left-most occurrence the defining occurrence. However, this leads to two problems. First, this would implicitly define an order for the reactants and make it impossible to rearrange them later on, for example for more efficient matching. The second (and maybe more immediate) problem is that there is no way to syntactically distinguish between definition and reference occurrences in a context-free parser. In fact, having access to the information whether a variable `x` has already occurred or not would make the parser context-sensitive[12].

In cases like this where the metamodel cannot be automatically inferred, Xtext allows disabling the automatic inference and working with a manually curated metamodel [21, pp. 358ff]. We used this feature to adapt the generated metamodel. In particular, we added the list of used variables as a new attribute `variables` to the reaction rule element, as well as a new metamodel class `Variable`. All variable occurrences anywhere in the

[12]This is comparable to the infamous context-sensitive C fragment `T * x;`, which can either be a pointer declaration or a multiplication, depending on whether T is a type or a variable. Also see en.wikipedia.org/wiki/Lexer_hack

rule are then references to a `Variable`. The variables are never explicitly declared in the concrete syntax. However, they could have been added to the abstract syntax, as it is done, for example, in *React*(C) [113]. The abstract and concrete syntax of ML-Rules leave the list of variables used in a rule implicit.

As the list of variables has no corresponding element in the concrete syntax, it can not be populated by the parser. Instead, the value has to be obtained after parsing. Xtext provides an extension point called `IDerivedStateComputer` that can be used for such post-processing. We use this extension point to collect all variable names used in variable references on the left rule side, eliminate duplicates, instantiate a `Variable` object for each of them, and point all corresponding references to them. The `Variable` objects are then assigned to the `variable` attribute of the reaction rule.

As a result, each variable is linked to all its occurrences and vice versa. The downstream code for type checking and code generation can work with a metamodel that contains a true representation of the rule.

6.4.3. Type checking

After defining the grammar rules and generating the metamodel and the parser from it, further code that relies on the metamodel can be written. One very important feature to implement is a type system, which consists of two different components: one for determining the type of a syntactical construct, and one for checking that the subconstructs of a syntactical construct have the correct type. The type checker can be implemented as a set of validation methods. These are invoked automatically when the user writes into the text editor, providing immediate feedback when entering expressions with type errors, for example as red “squiggly lines” [21, pp. 179ff.]. For example, the rate expression in ML-Rules must be of a numeric type. In the following rule, the rate results from multiplying a constant, an expression for the entity count, and a variable that was matched on the left rule side:

```
k: 1.0;

A(num);

A(x):a -> A(x - 1) @ k * (#a * x);
```

To make sure that the type of the rate expression is correct, the following steps need to be taken:

- The rate expression is a multiplication, which we defined as an operation that yields a numeric value. Thus, this check passes.
- The first argument of the multiplication is a constant. Constants are defined earlier in the model and their type can be inferred from their value. Obviously, `k` is a numeric constant and, thus, a valid argument for the multiplication.
- The second argument is itself a multiplication. With the same argument as above, we assume that it yields a numeric value and, thus, has the correct type.

- The first argument of the nested multiplication is `#a`, which is an expression that yields the count of the variable `a`. Thus, a numeric value and valid.
- The argument for the operator `#` must be a variable that was bound to a pattern (i.e., is of the type `ent`). Looking up `a`, we see that it occurred once on the left rule side. We can confirm that it was bound to the first and only pattern on the left rule side and, thus, is a valid argument for `#`.
- The second argument for the inner multiplication is the variable `x`. Again, we look up the variable and its only occurrence. It was bound as a pattern variable in the left rule side. To determine its type, we need to look up where it occurred, which reveals that it was used to match the first argument of an `A`. In turn, we look up the definition of the species `A` and find the type of its first attribute, which is indeed `num`, the type of numeric values. Thus, `x` is a valid attribute for the multiplication.

When one of the checks fails, the term that has the wrong type is underlined in the text editor generated by Xtext. The type of `x - 1`, the expression for the value of `A`'s attribute on the right rule side, can be checked similarly.

Apart from the patterns on the left rule side, the type checking is simpler than in GPLs. The language contains only a few simple types and no inheritance, polymorphism, generics, or other more advanced typing concepts. Due to the explicit types of arguments and the return value in function declarations, checking types at function call sites is not complicated as well.

6.4.4. Code generation

An ML-Rules model that is typed into the text editor generated by Xtext is continually parsed and validated (mainly type checked). If no errors are found and the model file is saved, Xtext's code generation component is invoked. The following Java classes are generated:

- A class `Model` as the entry point for instantiating a model. This includes all constants, the creation of the initial solution, as well as references to other classes as needed.
- An abstract class `AbstractFunctions` containing abstract declaration of each function declared in the model.
- For each species declaration, a class representing the species.
- For each rule, a class that contains a method for matching the rule to a solution and yielding the resulting reactions.

Before discussing the actual generation of pattern matching code for the rules, we shortly consider the integration of the generated code as listed above with handwritten code.

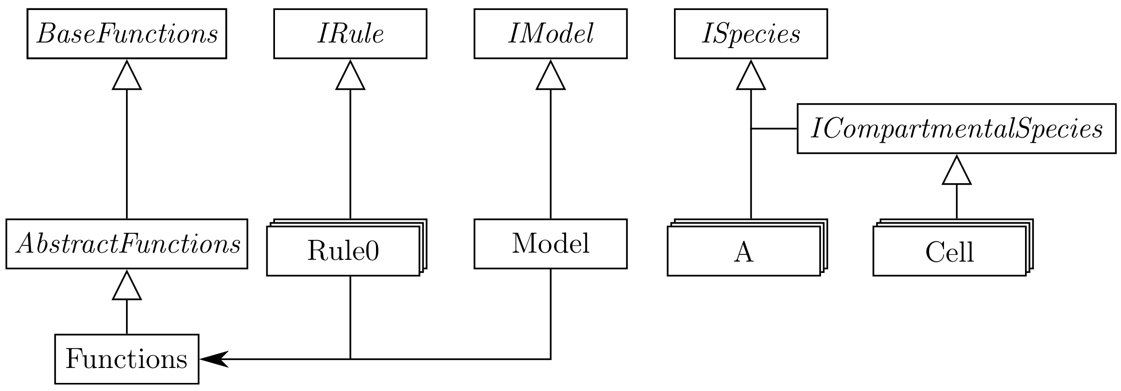


Figure 6.2.: A class diagram contrasting the handwritten and generated code. The uppermost classes are model-generic handwritten classes or interfaces. Below are model-specific generated classes. The class Functions is the only model-specific handwritten class.

Integration of generated code

The separation of generated and handwritten code is necessary because changes to generated code would be overwritten when the code is regenerated. Martin Fowler suggests a design pattern called "Generation Gap" to safely separate and combine both types of code [80, p. 571ff.]. The idea here is to define inheritance relations between handwritten and generated classes. For example, the generated class can inherit from a handwritten base class and in turn be the base class for a handwritten subclass. This way, a clear interface between generated and handwritten code is defined, and the generated code can be regenerated safely without interfering with the handwritten code (Figure 6.2).

The model-independent handwritten code is contained in a library (called "`mlrules3-common`") [13]. If the model contains functions on solutions, the generated class `AbstractFunctions` is abstract, and a subclass must be written to implement those functions. If the model contains only simple arithmetic functions or no functions at all, the generated class `AbstractFunctions` is not abstract and no subclass is necessary. The model-dependent handwritten code should also contain some execution entry point such as a class with a `main` method. For example, for a model with no user-defined functions, the following minimal `main` method executes the model with a simple simulation algorithm from the library:

```
Model model = new Model();
FirstReactionMethodSimulator simulator = new FirstReactionMethodSimulator(model);
simulator.run(10.0, 0.1); // stop time and observation interval
```

Again, this exemplifies how generated classes (`Model`) and handwritten classes (`FirstReactionMethodSimulator`) are integrated. The key point is the usage of a library

[13]To automatically include this library in ML-Rules Eclipse projects, we implemented a simple ML-Rules-specific "New Project" wizard.

with abstract, model-independent interfaces that abstract over the generated classes for concrete models. This enables the model-independent implementation of simulation algorithms that operate on the interfaces only. The library also provides further interfaces that are used in the abstract method declarations. For example, the interface `IRule` contains just one method declaration:

```
Iterable<IReaction> match(ISolution solution);
```

The types `IReaction` and `ISolution` are part of the library as well. Finally, the library contains a file with function declarations in ML-Rules syntax. These function definitions form a kind of standard library of functions, whose implementations are contained in the class `BaseFunctions` and inherited and usable for any concrete model. For example, we implemented the function `nu` that generates fresh link values in this standard library. This works without any customization as Xtext uses Java's class path concept to look up definitions, and the function declarations as well as their Java implementations are available via the `mlrules3-common` library.

Generating pattern matching code

ML-Rules' species and constants map nicely to Java classes and static final class members, and code for function declarations and definitions can be generated straightforwardly as well. In contrast, reaction rules can not be mapped directly to (object-oriented) Java concepts. Thus, the majority of the generation code is concerned with rules. The idea is to generate one class per rule, where each of those classes provides an implementation of the method `match` that can be applied to a solution and yields the instantiated reactions for this solution. The `match` implementation checks for each reactant in the rule and for each entity in the solution if the entity matches the reactant, while making sure that entities are not matched several times. Essentially, this is an implementation of pattern matching as described in Section 6.2.1. Generating this pattern matching code challenging in two ways. First, the code generation must handle any number of reactants of a rule and later matching steps depend on earlier ones. The generated code must account for the propagation of matching state similar to the nondeterministic state monad in the Scala implementation. Second, the code generates variables, which leads to the task of maintaining hygiene, that is avoiding name clashes when generating variable names [122].

The first issue is addressed by dividing the code generation works into four steps. First, the reactants are iterated as described in Section 6.2.1, which results in a list of code snippets that open blocks with a `{`.For example, `for`-statements are generated to implement the branching points in the tree in Figure 6.1 on page 94. Then, the snippet list is processed and the snippets are nested to create a indentation cascade of block openings. As a consequence, each matching step maps to a block of code that has in its scope the variables instantiated by the earlier, surrounding blocks. By accessing these variables, the dependence on earlier matching steps can be encoded. For example, when matching the second occurrence of a variable in a non-linear pattern the value assigned to the first occurrence is available and can be used for comparison. In the innermost block, which is entered for every complete match of the pattern, a reaction is created. Then, all the blocks are closed with correctly indented `}`. This way, we generate readable, nicely nested code.

The issue of naming hygiene can be tackled by exploiting the restrictions of ML-Rules' concrete syntax. To avoid clashes of generated names with variable names in the ML-Rules model, we exploit that names in ML-Rules are not allowed to contain the symbol $ and include that symbol in all generated names. To avoid clashes among generated names, we exploit the tree structure of the pattern on the left rule side and derive unique variable names from the corresponding position in the pattern. This way, variable names follow a predictable naming scheme, which we exploit to refer to earlier variables.

The code generated for pattern matching according to the rule `A[B(x):b + B(x) + rest ]:a -> A[rest] @ #b * (#b - 1);` is depicted as pseudocode in Figure 4. For each reactant, the available entities in the solution at hand are iterated in a loop (ll. 2, 8, and 17). This yields a structure of nested loops. The iteration variable is named after the position of the reactant on the left rule side: the first reactant becomes `$root$0`, the second one is `$root$1`, the first reactant in the subsolution of the first reactant becomes `$root$0$0` and so on. At each nesting level, these names are used to initialize uniquely named variables that hold the count of needed and available reactant entities (e.g., ll. 4 and 6). There are also several checks that lead to terminating an iteration early and starting the next one, for example if the entity does not belong to the reactant's species (e.g., l. 3) or if not enough entities are available (e.g., l. 7). When matching several reactants in one solution, the entities already matched are taken into account when determining the number of available entities (ll. 20–25). This realizes the counting of entities at the time of matching as discussed in Section 6.2.1. Further, for variables used in the model we generate variables that hold the corresponding values (ll. 14–16) or compare the current value to the one already found (l. 27).

In the innermost block, after all reactants have been found, the actual reaction object is instantiated and added to the list of reactions (l. 29). It consists of two closures that calculate the rate and execute the reaction effect, respectively. The code for these two methods can be generated straightforwardly by reusing the generated variable names. For example, for the rate expression `#b * (#b - 1)` the following expression is generated:

```
(available$b * (available$b - 1))
```

The variable `available$b` has been defined in an enclosing block (l. 15) and holds the count of `b` entities available in the solution `b` was matched in.

A more technical description of the generated code and the process of generating it is given in the Appendix B.3.

6.4.5. Discussion

Implementing ML-Rules as an external DSL allows freely designing the language's concrete syntax. In particular, this enabled implementing the language in a backwards compatible way referring to the existing ML-Rules 2 as presented in Section 4.1. Some of the choices for the concrete syntax would not be possible in most GPLs (and, therefore, internal DSLs). For example, ML-Rules 2 uses parentheses for plain attributes of entities, but

```
Input: a multiset solution$ with the contents of the solution
Output: a list rs$ of reactions
rs$ ← ∅
foreach $root$0 ∈ solution$ do
    if $root$0 does not belong to species A then continue
    needed$root$0 ← 1
    alreadyTaken$root$0 ← 0
    available$root$0 ← solution$.count($root$0) - alreadyTaken$root$0
    if available$root$0 < needed$root$0 then continue
    foreach $root$0$0 ∈ $root$0.subSol do
        if $root$0$0 does not belong to species B then continue
        needed$root$0$0 ← 1
        alreadyTaken$root$0$0 ← 0
        available$root$0$0 ← $root$0.subSol.count($root$0$0) - alreadyTaken$root$0$0
        if available$root$0$0 < needed$root$0$0 then continue
        b ← $root$0$0
        available$b ← available$root$0$0
        x ← $root$0$0.attribute_0
        foreach $root$0$1 ∈ $root$0.subSol do
            if $root$0$1 does not belong to species B then continue
            needed$root$0$1 ← 1
            if $root$0$0 = $root$0$1 then
                alreadyTaken$root$0$1 ← needed$root$0$0
            else
                alreadyTaken$root$0$1 ← 0
            end
            available$root$0$1 ← $root$0.subSol.count($root$0$1) -
             alreadyTaken$root$0$1
            if available$root$0$0 < needed$root$0$0 then continue
            if x ≠ $root$0$1.attribute_0 then continue
            rest ← $root$0.subSol - $root$0$0 - $root$0$1
            rs$ ← rs$ ∪ create new Reaction
        end
    end
end
```

Algorithm 4: The code generated to implement the rule `A[B(x):b + B(x) + rest]:a -> A[rest] @ #b * (#b - 1);` in pseudocode.

square brackets for encoding the subsolution. Similarly, the arrow symbol `->` and the `@` symbol can be used to separate reactants, products, and the rate expression[14]. More fundamentally, the semantics of binding variables in a pattern on the left rule side and using them on the right side and in the rate is a pattern that is not typically realizable in mainstream imperative or object-oriented GPLs[15]. The exemplifies how external DSLs allow integrating domain notations into the concrete syntax to a higher degree than internal DSLs.

In addition to the concrete syntax, we also defined a metamodel that represents the abstract syntax of ML-Rules. The explicitness of this abstraction is very useful in at least two ways. First, it facilitates checking the correctness of models beyond syntactical checks. For example, type checking or resolving references to variables benefit from the richness of the metamodel. Second, models can be manipulated more easily and safely on the metamodel level than on the concrete syntax level. Such manipulations can include simple refactorings in the editor, such as renaming a species or variable, but also conceptually challenging operations like model composition. An important prerequisite is the ability to generate the concrete syntax for a metamodel instance. Xtext provides this feature by default.

Using a language workbench like Xtext is a crucial ingredient to our implementation of ML-Rules as an external DSL. Xtext provides many editing features "for free" with minimal configuration or code. These features include undo/redo, syntax highlighting, and jump-to-definition. As ML-Rules 2 was not implemented with a language workbench, it either lacks these features or they had to be implemented manually. However, such features are an important ingredient for the user experience, in particular if users are used to IDEs. Nevertheless, the integration into the Eclipse ecosystem is a double-edged sword. On the one hand, users that are familiar with Eclipse will feel at home quickly and, for example, intuitively understand how the editor marks type errors. On the other hand, the Eclipse-based editor is a much heavier piece of software when compared to ML-Rules 2, for example considering the start-up time.

Another aspect that benefits from Xtext's defaults is the integrated generation of code. Each time a syntactically and semantically correct ML-Rules model is saved in the editor, Java code for the model is generated and immediately compiled. If necessary, the generated code can be inspected and, for example, break points can be set to debug the model behavior. The interaction with user-written code is exemplified by our handling of functions. Whereas functions that only wrap arithmetic expressions can be defined directly in the model, more complex functions on solutions that operate on entities and their attributes can be written in Java. Here, the generated Java classes for each species are available as well, allowing powerful functions that would be hard to implement with ML-Rules' syntax. The interface between generated and user-written code is expressed with inheritance relations. A particularly interesting consequence of this handling of functions is the possibility to ship a standard library of functions with the language.

[14] Some GPLs are quite flexible when it comes to integrating such symbolic syntax. C++, for example, allows overloading the semantics of parentheses, square brackets, and the arrow symbol `->`.

[15] A notable exception is Rust, which supports (linear) pattern matching.

Instead of encoding predefined functions in the syntax and semantics of the language, they can be implemented in plain Java. The machinery for making the standard library available in the language is again provided by Xtext.

6.5. Evaluation

Evaluating different avenues for DSL development is not trivial [206]. There are different criteria to consider when assessing a DSL implementation, including ease of implementation, ease of usage, computational efficiency, or extensibility. Quantitative data about usability can be obtained in user studies (which has been done for ML-Rules [123]). Instead of using a fixed scheme with weighted evaluation categories, we evaluate our implementations by comparing them to each other, to the preexisting implementation ML-Rules 2, as well as to implementations of other modeling languages, which we introduced in Section 4.2. To keep this section focused, we concentrate on a few core aspects.

6.5.1. Concrete syntax

The ability to freely design the concrete syntax is one of the main strengths of external DSLs, whereas the syntactical constraints are one of the main drawbacks of internal DSLs. Our implementations exemplify these respective characteristics. In particular, with the external DSL implementation we were able to replicate the syntax of ML-Rules 2 and get reasonably close to the abstract syntax. The less succinct internal DSL, on the other hand, combines Scala's for-comprehensions with domain-specific syntax for matching reactants.

The difference in syntactical design freedom is most notable in rule definitions and, to a lesser extent, in species declarations. Other aspects of the language, for example the definition of constants, is comparable in both implementations. Regarding function definitions the internal DSL even offers more powerful syntax as all features of Scala are available, including imperative and functional programming patterns. In the external DSL, the full power of Java is available, but only outside the actual model file.

One noteworthy aspect is that our internal DSL implementation does not require any syntactical escaping of expressions or quoting of names. This is often necessary in internal DSLs that do not directly reuse their host language for expressions or types. For example, in Chromar expressions for attribute values as well as rates must be enclosed in single quotation marks `''`[16]:

```
let r = [rule| A{x=x} --> A{x='x+1'} @'x' |]
```

Similarly, in PySB the names of species and attributes must be quoted when first used[17]:

```
Monomer('L', ['s'])
```

In contrast, our internal DSL implementation reuses Scala's expressions for attributes and rates as well as Scala's case classes for species. Therefore, no quoting or escaping is

[16]Snippet taken from github.com/azardilis/Chromar
[17]Snippet taken from pysb.org/

required. This allows better integration with existing editors, which often provide live detection of type errors or typos.

6.5.2. Editor support

All languages discussed in this thesis are textual and can be used with any text editor. However, the user experience in code editors is a crucial aspect of language engineering. Recent developments like the language server protocol (LSP) facilitate the implementation of powerful language support in editors, including support for DSLs [40]. Nevertheless, mainstream GPLs typically offer more sophisticated editor features than DSLs. The reason is, of course, that domain-specific languages also require domain-specific rather than off-the-shelf editor features. Implementing those features is sometimes not justified for short-lived DSLs with a small target audience.

Projects like language workbenches or, more recently, the LSP provide support for editor features when implementing external DSLs. With Xtext's default settings, for example, many editor features are added automatically. Therefore, the Xtext editor generated from the grammar without any customizations already surpasses the editor for ML-Rules 2 (called the "Sandbox") in terms of editor features. Xtext also provides extension points for more advanced features, for example auto-formatting the syntax. On the other hand, the ML-Rules 2 Sandbox also provides graphical interfaces for configuring and executing a simulation run with the opened model as well as displaying and exploring the results. Due to Xtext's integration into the Eclipse framework, these are not as straightforward to add in our Xtext-based implementation. It is possible, however, as shown by the BioNetGen GUI RuleBender[18]. RuleBender is an Eclipse-integrated rule-based modeling editor with additional graphical interfaces [216]. As it uses ANTLR instead of Xtext for parsing, editor features such as auto-completion are not available.

For internal DSLs the host language's editor can be reused, and all its features should to some degree be usable for working with the DSL. For example, species in our Scala DSL are encoded by case classes, which can be safely renamed by namespace-aware editors like IntelliJ IDEA. Mouse-over type hints, or jump-to-definition are supported as well.

In summary, both our external DSL implementations provide editor support in different ways. In the internal DSL all editor support is provided by editors for the host language "for free", but the support is not easy to modify or extend. External DSL editors that are implemented as standalone applications (like the ML-Rules 2 Sandbox) have the problem that editor features have to be implemented by hand, which is cumbersome. This problem is alleviated when using language workbenches like Xtext.

6.5.3. Computational efficiency

Computational efficiency is always an important topic for simulation applications. How long a simulation run takes depends, of course, mainly on the simulation algorithm, and different algorithms for ML-Rules have been proposed [101]. Therefore, the language implementation's impact on efficiency can be considered under two aspects. First, how

[18] github.com/RuleWorld/rulebender

does the language implementation affect the ability to select an algorithm? And second, how does the language implementation affect the efficiency of a given algorithm implementation?

Selecting a simulation algorithm

In Section 6.2.3 we shortly discussed network-based simulation as an alternative to the network-free simulation algorithms that we mainly use in this thesis. Network-based simulation is not applicable to all rule-based models. When applicable, however, network-based simulation is often more efficient than network-free simulation. To determine whether a network-based approach is feasible, it can be helpful to inspect the rule-based model via static analysis. Does it contain species with unbounded attributes? Does it contain rules that create new compartments? A positive answer to one of these questions indicates that the reaction network might be infinite and that a network-based simulation algorithm is contraindicated. In our Scala-based internal DSL, however, these questions can not be easily answered. The required information is not available, as the corresponding language elements are handled by the host language. In our external DSL, on the other hand, the needed information can be easily obtained from the metamodel after parsing by static analysis.

Another example is the application of partial propensity variants of the SSA [194, 174]. These efficient algorithms pose specific requirements on the form of all reactions. In particular, partial propensity algorithms only support rate expressions in which the population size of each reactant is a factor. However, ML-Rules also allows other types of rate expressions. Therefore, the reactions and rate expressions of a given ML-Rules model must be checked for adherence to the requirements of partial propensity algorithms before such an algorithm can be applied. Again, this is easy in the external DSL and impossible[19] in the internal DSL. A third example is the automated derivation of a dependency graph for the next reaction method (Section 3.2.2). To generate such a dependency graph, reaction rules need to be analyzed to infer which information they use to generate reactions.

It should not be concluded from the above points that model analysis is only possible in external DSLs. For example, PySB is an deeply embedded internal DSL and compatible with BioNetGen and its network-based simulation algorithm. The decisive point is the type of embedding that a given language uses (see Section 2.3.3). Our internal DSL relies on a shallow embedding to directly expose the pattern matching algorithm as a monadic computation, without an explicit representation of the abstract syntax as in deeply embedded internal DSLs. In contrast, the external DSL creates an intermediate representation (metamodel) of a ML-Rules model that corresponds to the abstract syntax of ML-Rules. This metamodel is then used to generate code that implements the pattern matching.

The ML-Rules 2 implementation is an external DSL as well, but does not use a dedicated metamodel or an explicit AST. Instead, it operates directly on the parse tree. This makes

[19]Impossible at least without fundamentally changing the language or relying on metaprogramming.

analyses of the model like the examples above harder than in a metamodel because, for example, different occurrences of the same variable are not linked to each other.

In summary, inferring what simulation algorithm is applicable for a given model requires insights into the model definition. Deeply embedded languages allow making an explicit model definition available, preferably in a metamodel. In contrast, shallow embeddings, in particular when reusing the host languages' syntax, tend to lose information about the model, making the choice of the simulation algorithm harder.

Efficient implementation of simulation algorithms

Even when the same algorithm is used, different approaches for implementing a language can lead to vastly different performance. Much of the difference is caused by the difference between compilation and interpretation, as described in Section 2.4.1. In the context of DSLs, compilation almost always means generating GPL code (see [140] for an example of directly generating assembly code). Whereas the code generation method relies on compiling a hard-coded model-specific algorithm implementation, the interpretation approach uses a generic algorithm that interprets the model as part of its input. In particular in the context of simulation applications, the compilation approach has the opportunity "to eliminate all interpretive overhead" and, thus, increase performance significantly [117]. Similar results have been obtained regarding internal DSLs [36]. Kameyama et al. even claim that "code generation is the leading approach to making high-performance software reusable" [115].

Nevertheless, interpretation is often the simpler implementation strategy. In particular, all code is executed in a single execution state, which is easier to reason about than multi-stage programs. This is the approach taken by ML-Rules 2. Here, the program parses a model file into an in-memory parse tree and then traverses this parse tree each time it matches a rule against a solution, calculates a reaction rate, or evaluates a function. In other words, the source code is completely generic and model-specific source code never exists.

Both of our two new implementations take a different approach. In the internal DSL, the model is written in Scala and co-exists with the generic code, for example for simulation algorithms. Consequently, there is only one execution stage and model and simulator can interact directly. The model-specific source code in the external DSL, on the other hand, is generated automatically. Therefore, the implementation is split into two execution stages: code generation and, after the generated code is compiled, execution. As described in Section 6.4.4, generating error-free code and integrating it with handwritten code is not trivial—in particular compared to the internal language implementation, in which model-specific and generic code is compiled in one stage.

One way to reap the benefits of code generation without needing to adapt the software architecture is runtime code generation. In particular for languages that run on a virtual machine, adding code for execution is often possible even at runtime. For example, ML-Rules 2 includes a feature that allows replacing subtrees of the parse tree with tailored nodes compiled at runtime [154]. This can be applied easily to rate expressions, where instead of interpreting a subtree with one node for each arithmetic expression,

Simulator	Run time
Java Reaction-based generic	0.396 s
Java Reaction-based specialized	0.234 s
Java Reaction-based specialized without objects	0.153 s
C Reaction-based specialized	0.121 s

Table 6.1.: Runtime of the Wnt model in different implementations

some Java code that directly uses Java's built-in arithmetic operators is compiled. By avoiding indirections, for example due to following object pointers, the evaluation time for arithmetic expressions could be reduced by over 80%, and overall simulation time by up to 40%.

Another source of indirection is the mapping of species and entities to code. In model-specific source code, species map naturally to types and entities to values of these types. This mapping allows seamlessly working with entities. For example, in the internal DSL this enabled us to use Scala's pattern matching; in the external DSL we exploited Java's object-oriented concepts like inheritance. Also, the attributes of an entity and their types are represented directly in the language. In contrast, the source code of an interpreter typically contains a type that represents all species and each actual species in the model is represented by a value of this type. The attributes of an entity and their types are handcoded, and typechecking must be implemented manually. Thus, model-specific source code allows reusing more host language concepts than interpreters, which can also give the host language compiler more opportunities to optimize the program.

To illustrate the potential of partial evaluation and code generation, we conducted a simple performance experiment with the Wnt model, implemented in handwritten code with different levels of specialization (Table 6.1). All implementations use the same SSA variant (the optimized direct method [42], also see Section 3.2.2) for a reaction-based formulation of the model as shown in Section 4.1. The model is executed for 720 time units. Starting with a generic simulation algorithm in Java that receives the model as input, we first specialized the algorithm to the model and then streamlined the code further by eliminating all object allocations, leaving only stack-local primitive variables. This last version could be easily ported to plain C, which was then compiled with maximum optimizations (the `-O3` flag for `gcc`). The results show that by partial evaluation, without changing the simulation algorithm or programming language, speed-up factors of more than 2 can be achieved. This speed-up is achieved purely by specializing the simulation algorithm to the model. Switching from Java, a VM-based language with a just-in-time-compiler to the ahead-of-time-compiled C gave another speed boost[20]. This shows that optimizing compilers can be a valuable tool when model-specific source code is available [124].

To summarize this section on performance implications of DSL implementation techniques, we can state that deeply rather than shallowly embedded languages enable computational efficiency. First, they allow statically analyzing the model in-depth and

[20]This speed-up could also be caused by the switch to another PRNG.

selecting a simulation algorithm based on the analysis results. Second, deep embeddings like the metamodels in Xtext are well-suited for generating efficient model-specific code. Model-specific and generic code is then compiled together, allowing whole-program optimization, which is an avenue to highly efficient programs (see Shivkumar, Murphy, and Ziarek [211] for a recent example).

6.5.4. Extensibility and interoperability

As a last point of comparison we take a look at how well the language implementations can be extended or combined with other software. Both aspects are important. The extensibility of a language is crucial to react to changing requirements. In particular in scientific research, simulation modeling languages are constantly challenged by the application domain to increase their expressive power or execute more complex models faster. Interoperability is crucial to integrate simulation runs into a simulation study. This can range from using a plotting tool to visualize the results of a single run to complex simulation experiments (see also Section 7.1).

The internal DSL has the big advantage that the model and the simulation algorithm are written in Scala. Therefore, plain Scala code can be used to add new features directly to the modeling language, even while working on a specific model. For example, if a specific piece of code occurs multiple times in a model, it may be factored out into a function definition. This way, the internal DSL allows rapid prototyping by editing a model, the pattern matching, and the simulation algorithm simultaneously. Being based on Scala, the internal DSL also facilitates integration with libraries from the Scala and Java ecosystem. Calls to these libraries can be easily integrated into the model itself (for example, to calculate reaction rates based on library functions or constants) as well as into code around the individual simulation runs, for example to analyze the model outputs. PySB is another example for an internal DSL that allows exploiting the library ecosystem of its host language, in this case Python.

The integration of simulation runs into other software is easy in the external DSL as well. After all, a simulation run is executed by hand-written Java code, and other Java libraries can be naturally invoked here. However, it is not that easy to use libraries in the model, as the model itself is not written in Java. For the same reason, it is much more difficult to develop new language features in the Xtext-based implementation, especially when (as in our case) the metamodel can not be automatically generated from the grammar. A change in the language now includes adapting the metamodel, the grammar, the code generator, and (potentially) the type checker, interspersed with generating code. Thus, development of an external DSL such as our Xtext-based language happens much slower.

As an example for the different degrees of extensibility in both implementations, consider modifying the pattern matching algorithm in such a way that count expressions $\#x$ now yield the total count of x entities instead of the count at the time of matching. In the internal DSL, the modeler can redefine the value `solution` (see page 103) directly in the model file to implement this change. In the external DSL, however, this modification can not be done in the model. Instead, the code generator in the Xtext project needs to be adapted and the modeling environment must be restarted.

6.5.5. Summary

One way to summarize this evaluation is saying that our deeply embedded, external DSL has more potential as a modeling language than the internal, shallowly embedded DSL, but also requires more implementation effort. It allows more freedom in designing the language syntax, more efficient simulation algorithms, and more domain-specific tooling. Language workbenches like Xtext support the implementation of external DSLs, for example by making the abstract syntax explicitly available as a metamodel. But even then, many language aspects have to be implemented manually. The shallowly embedded internal DSL, on the other hand, reuses the syntax of the host language and omits representing the abstract syntax. This makes the language implementation very succinct and easy to modify, but also limits its possibilities, for example in terms of static analysis of models.

6.6. Excursus: Continuous-time agent-based modeling in social science

So far we have seen how population-based processes can be modeled as Markov Population Models (see Section 3.2), a class of CTMCs. To express models succinctly, we employed the modeling paradigm of rule-based modeling. Now we transfer the idea of CTMC-based modeling to another modeling paradigm, the area of agent-based modeling. Our main contribution here is integrating principled continuous-time discrete-event simulation into an existing software framework for agent-based modeling and simulation. Those well-established frameworks provide many practical features for running simulations, which can then naturally be combined with the CTMC modeling approach.

Some processes in social science can not be modeled in a population-based manner. Here, the paradigm of *microsimulation* has emerged, which considers each member of the population on its own, having an individual state and an individual view of the world [121]. Traditionally, microsimulation models contain a very abstract representation of decisions and "are not very explicit and detailed about the path the [...] subjects follow to reach a decision" [121, p. 33]. Agent-based modeling (ABM) is a subdiscipline of microsimulation that models the internal processes of individuals in more detail.

Diverse software packages support scientists in developing and using agent-based models [193]. Some notable examples for such software are Repast Simphony [168], MASON [145], and NetLogo [251]. These simulation tools typically offer a simple way to schedule events repeatedly at equidistant time points. Thus, they encourage the simulation paradigm of *discrete time step simulation*, where the model changes its state at fixed points in time. Additionally, events can also be manually scheduled at arbitrary points in continuous time, allowing for *discrete event simulation* (in NetLogo, this requires an extension [210]).

The step-wise schedule, however, is used by the vast majority of agent-based models that are implemented in these frameworks [193]. For example, the influential "Wedding Doughnut" model [213] implemented in Repast Simphony and the MASON RebeLand model [51] use a step-wise approach. This observation is in stark contrast with inclinations in social science to use models with continuous time-bases. For example, demographers consider continuous-time models superior to discrete-time models, as they are more precise and allow for the integration of established domain-specific analysis methods [253].

A possible explanation for the divergence between the step-wise and the event-based approach might be the easier mapping of available data to a step-wise model. For example, a typical data set in demography might contain the percentage of deaths in each age cohort per year. Then an agent's death probability can be directly inferred from its age in a simulation with a step size of one year. Similarly, the probabilities for other demographic events such as marriage and childbirth can be estimated from data. However, such a step-wise approach effectively models the data set. To instead build a model of the underlying, data-generating processes, a discrete-event approach is more adequate, as individual decisions and events can not be assumed to happen equidistantly in time [253].

Thus, in modeling processes in social science, the waiting time in between events is of central interest. If these waiting times are exponentially distributed, the model is a CTMC. For some processes in social science, population-based CTMC as presented in Section 3.2 models suffice. Those can be modeled similarly as the biochemical systems we already saw by identifying populations and defining state transitions that modify the populations sizes. An example for modeling population dynamics with CTMCs in social science are SIR epidemic models, where individuals change between susceptible (S), infectious (I), and recovered (R) sub-populations [4]. Then, a state can be represented by a triple (S, I, R) of the sub-population sizes.

To represent the individuals in the model in more detail, they can be distributed into more sub-populations. For example, additionally distinguishing individuals by sex yields 6 populations (SF, IF, RF, SM, IM, RM). Adding, for example, 10 different age cohorts means that every state must track 60 sub-populations. Obviously, increasing the number of attributes leads to a combinatorial explosion of the number of distinguishable populations. This is the same effect that motivated the introduction of rule-based modeling in Section 4.1.2. And similarly, individuals equipped with at least one continuous attribute lead to an infinite number of populations in a single model state. It is reasonable to assume that agent-based models in social science require even more attributes than in this example. Thus, sorting agents into populations is futile.

In addition, the assumptions of a "well-stirred solution" is often not justifiable. Instead of modeling individuals as indistinguishably equal entities in a population, they are represented as nodes in a graph, with edges representing social ties. This leads to the metaphor of individuals having "linked lives", where the decisions of each individual agent in the model depend on its attributes as well as its unique locations in a graph of social links. Of course, this makes the model's state space even more complex. But such a system can still be formulated as a CTMC, by letting the different possible behaviors of the agents compete in a stochastic race. This way, complex agent behavior such as decision processes can be modeled [244, 118].

To describe such a model, the ideas discussed in Section 3.2.1 can be reused. We can provide a succinct, finite description of a potentially infinite model by using an appropriate modeling formalism/DSL. In particular, concepts of rule-based modeling languages for population-based CTMCs can be transferred to the agent-based setting. Each rule specifies a potentially infinite class of state transitions. However, to apply the ideas of rule-based modeling to agent-based models, some of the concepts have to be adapted. In the following, we describe how a rule-based DSL can be integrated into an existing framework for agent-based modeling, exemplified by Repast Simphony [168].

6.6.1. Adapting the simulation algorithm

First we consider the effect of agent-based models on the stochastic simulation algorithms as presented in Section 3.1.4. In comparison to population-based models with a small number of populations, each state in an agent-based model has a large number of possible transitions[21] to a successor state. The reason is that, as each individual can be distinguished in the current state, we also have to distinguish which individuals are actually changed by the state transition, as that leads to distinguishable successor states. Thus, the number of state transitions is large, which affects the efficiency of the First Reaction Method and the Direct Method. These algorithms have to recalculate the rate for every possible transition at every step, which takes more time if more transitions have to be considered.

As an alternative, we can adopt the idea of the Next Reaction Method (Section 3.2.2). Here, a dependency graph is used to avoid updating rates that are unchanged. With agents placed in a graph, transition can be expected to have only local effects and leave many rates unaffected. Thus, the efficiency of the simulation algorithm does not linearly increase with the number of agents anymore, but more slowly (depending on the connectivity of the graph) [196].

The Next Reaction Method relies on managing events in a priority queue where the priority of an event is its time stamp [38]. Event queues are employed by many other simulation paradigms (see Section 6.1.1), where events are added to the queue or retracted from the queue. In addition, the time stamp of an event may be modified and the event rescheduled accordingly. This operation is used, for example, by DEVS simulation algorithms, where the overall number of events is constant, as one event is associated with each model component. The encapsulation of DEVS model components leads to locality of state, which can be exploited by only rescheduling events if necessary. The Next Reaction Method uses the same idea. In conclusion, the fundamental idea of efficient simulation for rule-based agent-based models is exploiting locality.

6.6.2. Implementing continuous-time simulation in frameworks for agent-based modeling

In the previous section we have established that continuous-time agent-based models can be implemented by keeping future events in an event queue. Before presenting a rule-based approach to specify such models, we investigate how scheduling events in continuous time

[21] In contrast to the biochemical population-based case, we can not call these state transitions “reactions” anymore.

is supported in state-of-the-art ABM frameworks. We focus on Repast Simphony here, but similar points can be made for MASON and NetLogo.

In Repast Simphony, which is implemented in an object-oriented manner in Java, the event queue is represented as an explicit schedule object. This schedule object allows explicitly scheduling and retracting events. More frequently, Repast models use methods of the schedule that cause calls to methods on agent objects at fixed time points or a fixed interval. Alternatively, agent methods can be equipped with Java annotations that have the same effect. This enables a very succinct implementation of simulation with fixed time steps.

To implement discrete-event simulation with ABM frameworks, events have to be scheduled manually. Essentially, this corresponds to the event-scheduling world view described in Section 6.1.1. However, little work exists that shows how to do this for exponentially distributed waiting times. There is a tutorial that walks through implementing a discrete-event queuing system in Repast Simphony [223]. For NetLogo, an extension exists that allows for manual event scheduling [210]. To illustrate the necessary steps, we present an implementation of an agent-based SIR model in Repast Simphony.

The model contains a network of agents, with each agent having its infection state as an attribute. Following the SIR approach, the possible values for the infection state are susceptible, infectious, and recovered. The network links between agents represent social contacts and do not change during the simulation. To initialize the model, a number of agents is created and the agents are randomly linked. A proportion of the agents is initially infectious, the remaining ones are susceptible.

Two types of events are considered.

Infection A susceptible agent a_s becomes infectious with a rate $a \cdot |\{i | i \in \mathit{neighbors}(a_s), i \text{ } \mathit{is} \text{ } \mathit{infectious}\}|$, where a is a rate constant. For an agent without infectious neighbors, this rate is zero and no infection event will be scheduled.

Recovery An infectious agent recovers with a constant rate b.

Besides the rate constants a and b, the model is parameterized with the total number of agents in the model, the number or proportion of initially infectious agents, and a method the generate the random network.

At most one event is scheduled per agent. For susceptible agents without infectious neighbors or recovered agents no event is scheduled. The recovery of infectious agents does not depend on their network neighbors and, thus, must never be rescheduled. In contrast, the rate of the infection of a susceptible agent depends on the number of infectious network neighbors. Thus, infection events need to be rescheduled when the number of infectious network neighbors changes due to a neighbor getting infected or recovering. This leads to the event scheduling logic shown in Figure 6.3. An agent that gets infected or recovers informs its neighbors, which reschedule their infection event, if present. All other events in the schedule remain unchanged. This way, locality is exploited to avoid all unnecessary rescheduling.

```java
 1 public class Agent {
 2
 3   /* ... */
 4
 5   private ISchedulableAction scheduledEvent;
 6
 7   public void getInfected() {
 8     this.infectionState = InfectionState.INFECTIOUS;
 9     scheduleRecovery();
10     informNeighbours();
11   }
12
13   private void informNeighbours() {
14     for (Agent agent : network.getAdjacent(this)) {
15       agent.rescheduleInfectionEventIfPresent();
16     }
17   }
18
19   public void rescheduleInfectionEventIfPresent() {
20     if (infectionState == InfectionState.SUSCEPTIBLE) {
21       if(scheduledEvent != null) {
22         schedule.removeAction(scheduledEvent);
23       }
24       scheduleInfection();
25     }
26   }
27
28   private void scheduleInfection() {
29     double currentTime = schedule.getTickCount();
30     double infectiousNeighbors = getInfectiousNeighbors();
31     if (infectiousNeighbors == 0.0) {
32       scheduledEvent = null;
33     } else {
34       double rate = infectionRate * infectiousNeighbors;
35       double waitingTime = RandomHelper.createExponential(rate).nextDouble();
36       scheduledEvent = schedule.schedule(
37         ScheduleParameters.createOneTime(currentTime + waitingTime), this, "getInfected");
38     }
39   }
40 }
```

Figure 6.3.: Java snippet from an agent-based SIR model implemented in Repast Simphony. Agents keep track of their next event (line 5). Upon infection, an agent informs its neighbors (line 10). The susceptible neighbors then retract the existing infection event (line 22), reevaluate the infection rate (line 34), sample a new waiting time (line 35), and schedule a new infection event (line 36). Figure taken from Warnke, Reinhardt, and Uhrmacher [241].

6.6.3. Rule-based modeling

Whereas the manual scheduling as shown in the previous section is a valid approach to implementing agent-based models, it has the problem of interleaving model and simulation code. This conflicts with striving for a separation of concerns between these two components. One way to factor out the model description is designing a DSL in which the behavior of the agents is specified in a rule-based manner. A single generic simulation algorithm can then be used to execute models described in this DSL. Thus, the idea is the same as for ML-Rules and similar languages.

To apply this idea to ABM frameworks, we must address two key challenges.

- We need to integrate the DSL into the ABM framework Repast Simphony. This way, the existing features of ABM frameworks can be applied to models defined in a rule-based manner. These features include the GUI, visualizations, or analysis tools.
- As discussed in Section 6.6.1, it is important to exploit locality of event effects. Thus, the simulation algorithm must be able to make assumptions about the dependencies between events. This should lead to a simulation scheme similar to the manually designed one in Section 6.6.2

The challenge of integrating a DSL into Repast Simphony can be addressed by defining an internal DSL in Java. We rely on a few concepts of the object-oriented programming paradigm to define an interface between reusable and model-specific components. The *simulation layer* contains the reusable part including simulation algorithms and utility functions (see Figure 6.4). It also exposes abstract Java classes and interfaces that are inherited from when specifying a model with the DSL. First, the abstract class `Agent` is the base class for all agents in the model. For each agent type in a model, an `Agent` subclass is implemented that contains the agent type's attributes as members. Additionally, the `Agent` interface allows adding instances of the `Rule` interface. A `Rule` object implements three methods that specify guards (which agents does this rule apply to), the waiting time (potentially stochastic), and the effect of executing the rule. These methods can be succinctly implemented as lambda expressions as introduced by Java 8 directly in the agent class and, thus, access the agent's attributes.

Using the DSL exposed by the simulation layer, the behavior of the agents in the SIR model from Section 6.6.2 can be expressed as follows:

```
public class SIRAgent extends Agent {

/* ... */

addRule(  () -> this.isInfectious(),
          () -> exp(recoverRate),
          () -> this.infectionState = InfectionState.RECOVERED);

addRule(  () -> this.isSusceptible(),
```

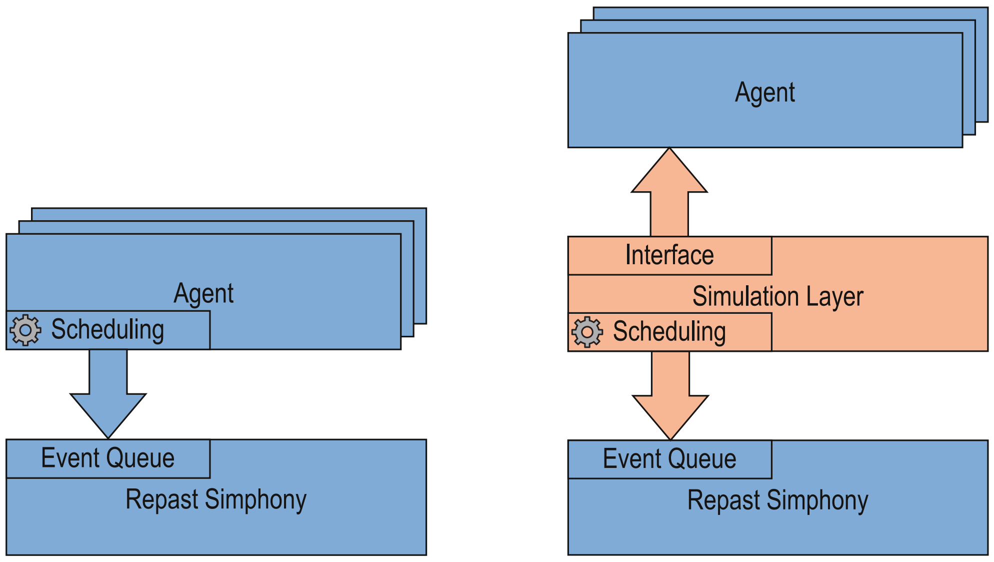


Figure 6.4.: Defining a Repast Simphony model with manual scheduling (left) and with the simulation layer (right). When scheduling manually, each agent class accessed the event queue directly to schedule and retract events. With the simulation layer, agent classes do not reference the event queue directly, but implement an interface that allows the simulation layer to query them. All scheduling is then done by the simulation layer.

```
        () -> exp(infectionRate * neighbours(this).
                filter((SIRAgent agent) -> agent.isInfectious()).size()),
        () -> this.infectionState = InfectionState.INFECTIOUS);
}
```

The `addRule` method is part of the abstract class `Agent`. It takes three arguments for the guards, the waiting time, and the effect. The first rule represents the recovery of an infectious agent, while the second rule models the infection of a susceptible agent. In the rule definitions all attributes and methods of the agent are accessible. Note that these rules do not express dependencies between events or talk about the machinery of when and how events are rescheduled.

We implemented the First Reaction Method and the Next Reaction Method in the simulation layer. Both algorithms can use the same agent definitions as their input, demonstrating the independence of model and simulator. The algorithms initially query all agents for their rules, evaluate them, and schedule events accordingly (see Figure 6.5). However, the simulation layer does not schedule the execution of the actual rule effect, but instead schedules a call to its own `executeEvent` method with the agent and the rule effect as arguments. Thus, `executeEvent` gets executed for each scheduled effect. Besides executing the effect, both algorithms handle events in this method. The First Reaction

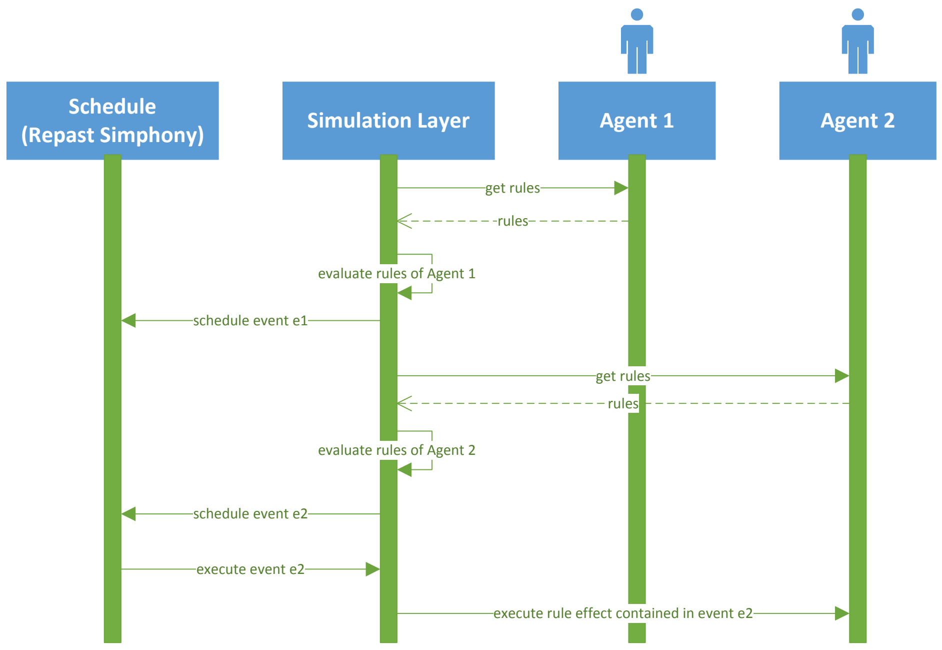

Schedule (Repast Simphony)
Simulation Layer
Agent 1
Agent 2
get rules
rules
evaluate rules of Agent 1
schedule event e1
get rules
rules
evaluate rules of Agent 2
schedule event e2
execute event e2
execute rule effect contained in event e2

Figure 6.5.: Sequence diagram illustrating the communication between the Repast Simphony core, the simulation layer and the agents in the Next Reaction Method. The agents and the schedule are not directly communicating.

Method queries all agents again for their rules, evaluate the resulting state transitions, and selects the state transition with the shortest waiting time as the next event to execute (stochastic race). In contrast, the Next Reaction Method queries only the agents that are affected by the effect and reschedules only these events accordingly.

To determine which agents are affected, the dependencies between the executed effect and the conditions and rate expressions of all rules need to be considered. In our implementation of the Next Reaction Method, we assume that the execution of an effect affects all immediate network neighbors and the acting agent itself. This way, we overestimate the dependencies in the SIR model. For example, recovery events (which are completely independent of network neighbors) are rescheduled as well. In other models, however, events can affect other agents over more than one network hop. A precise analysis of the range of effects would be necessary to fully exploit locality. Some work in that area exists already.

One example for an approach that exploits locality and dependency information is the Next Subvolume Method for biological reaction-diffusion systems [65]. Here, space is partitioned into connected subvolumes. Each subvolume contains populations of entities, which can react with each other. Such a reaction inside a subvolume only affects the populations in that subvolume, and reactions of populations in other subvolumes are unaffected. The other type of event is a diffusion of an entity from one subvolume to another, which then affects only the populations of the source and the target subvolume.

This way, a global schedule containing events of all subvolumes can be maintained and updates be kept to a minimum by exploiting locality.

Exploiting locality in agent-based simulation has been suggested in the context of distributed simulation [229]. Here, "spheres of influence" represent the part of the system state that is affected by the execution of an event. To minimize communication between computing nodes and, thus, increase performance, highly interdependent agents can then be clustered on one node. The Global-Scale Agent Model is another example for exploiting locality in agent-based modeling [179]. In this epidemiological model only an active subset of the overall population of agents is considered by the simulation. With infectious agents being active, new agents can only become active (= infectious) due to contact with an active agent. Thus, inactive agents that are not close to active ones can be disregarded by the simulation algorithm.

Integrating a more precise way to determine dependencies into the simulation layer, for example by static analysis, is subject to future work (see Section 8.3).

6.6.4. Discussion

In this section, we have shown how the techniques to model and simulation CTMCs of population processes can be applied to agent-based models. As all agents are distinguishable individuals, the number of state variables and state transitions grows much quicker than in population-based models. One way to still be able to execute agent-based models efficiently is exploiting that state transitions only change a small part of the state. Consequently, most state transitions that were possible before can remain unchanged.

The language in which the model is expressed is different from ML-Rules. In particular, there is no pattern matching and not even a left rule side. Instead, the rules in our agent-based DSL are inspired by stochastic guarded commands [103], as they have also been adopted by the agent-based modeling language ML3 [244]. Here, rules are assigned to agent types and define constraints and a rate expression based on the agent's individual attributes and neighborhood. The rule effect is then a sequence of mutations of the model state from the perspective of the acting agent, allowing it to change its own attributes and local surroundings.

It is valid to ask whether this style of modeling can still be called "rule-based". There is no declarative before-after transition as in ML-Rules and similar languages with reaction rules. More technically, there is also no binding of variables on the left side over which expressions can be formed. Instead, the presented DSL follows an object-oriented, individual-centric approach where attributes and network neighbors are retrieved by operations on the `this` reference to the acting agent itself. As such, it can be seen as an adaptation of rule-based modeling for individual-based modeling. Thus, the contribution of this DSL is lifting continuous-time agent-based modeling from the event-scheduling to the more abstract process-interact world view (cf. Section 6.1.1).

The more abstract level of model description allows us to express precisely the model behavior. In particular, we do not need to handle scheduling manually, which heavily obscured the actual model in our initial implementation with manual event scheduling. This is another instance of the separation of concerns between model and simulator. Again, the key to this separation is a DSL that acts as an interface to a reusable algorithm. By reusing the existing ABM framework Repast Simphony, the algorithm itself is only a thin layer that translates between the model and the framework. For the framework, the model behaves like any other model and, thus, all further features of the framework are applicable to visualize or analyze the model.

We can also draw conclusions in the other direction. One takeaway for implementing biochemical reaction networks could be to discard the population-based aggregation in favor of individual-based handling of entities. For instance, species with many possible attributes can be more efficiently simulated as individuals [105]. In particular compartmental species, which need to contain exactly the same subsolution to be aggregated in populations, would benefit from being handled individually. It is important to mention that this must be a purely algorithmic optimization and should not change the syntax or semantics of a language.

6.7. Summary

In this section we have presented two implementations of ML-Rules's semantics as proposed in Chapter 5, as well as one adaptation of CTMC semantics to agent-based modeling.

Our first implementation is based on functional programming and uses the internal DSL approach frequently found in functional programming. The DSL reuses the expressive host language Scala with its algebraic data types and associated pattern matching. It also employs monadic chaining to express the matching of subsequent patterns in the functional paradigm. As a consequence, the language implementation is very succinct, whereas models are comparatively verbose.

Whereas the first implementation adopts functional idioms, the second implementation follows object-oriented principles to create an external DSL with the language workbench Xtext. The core of the implementation is a metamodel, which is an object-oriented representation of the abstract syntax. Based on that metamodel, we wrote a Java code generator that allows executing a simulation run of an ML-Rules model. The generated code and the model-independent generic code interface via object-oriented inheritance relations.

The third implementation injects an internal DSL into the Java-based agent-based modeling framework Repast Simphony. This DSL has object-oriented and functional elements, as it operates on agents (which are Java objects), but encodes their behavior through Java's lambda expressions (a functional idiom). The utility of this DSL, however, is that it frees the agent-based model from execution-specific code and instead supplies an execution semantics based on the agents' behavior specification.

A few common aspects emerged in these three implementations. It appears reasonable to assume that these aspects are generally associated with DSLs for CTMC-based simulation modeling, in particular with rule-based modeling. We shortly summarize the commonalities and differences below.

- All three implementations allow expressing the rate of a state transition in dependence of the (re)acting entities. The two ML-Rules implementations make the reactant's attributes and multiplicities available, whereas the agent-based language gives access to the attributes of the acting agent and its network neighbors. The implementations differ technically. In the Scala-based internal DSL, we exploited Scala's pattern matching to capture values in variables. In the Xtext-based external DSL, the grammar allows non-linear patterns, which are appropriately encoded in the metamodel, and used to generate code. In the generated code, Java variables are defined in nested blocks, and the rate expression is then defined in the innermost block, closing over the variables. The agent-based language defines rate expressions as closures in the scope of the agent class, which makes the fields and methods of the class available. Thus, there are several effective ways to express and evaluate the dependence of a state transition on its context.

- Similarly to the rate expression, the effect of a reaction or behavior rule has to be encoded. How this is done also depends on how the state itself is represented. In the Scala-based internal DSL, the state is a tree structure of entity populations, and the effect caused by a reaction is encoded as a change vector. The other two implementations encode the state change as a closure over the determined context of the (re)action, mutating the state when evaluated.
- One recurring theme in all implementations is the need for static analyses. Especially the selection and configuration of the simulation algorithm can depend on such analyses. The agent-based language, for example, requires some kind of analysis to determine how executing an event affects other already scheduled events. Without automatic (static or dynamic) analyses, the user needs to provide a limit for the effect range. Of all three implementations, the Xtext-based internal DSL provides the most support for static analyses. Due to its explicit metamodel, predicates over the model code (mainly the reaction rules) can be expressed and evaluated. In addition, static analyses benefit from the rule-based nature of ML-Rules. Rule-based modeling languages with their declarative core syntax facilitate analyzing the model code. For example, the comparatively small syntax of ML-Rules allows inferring whether a model employs dynamic compartments. However, the presence of functions on solutions in an ML-Rules model, in particular user-defined functions, potentially eliminates this benefit. This again illustrates that increasing the expressiveness of a modeling language is associated with challenges for the simulation algorithm.

This concludes the chapter on implementing the CTMC-based modeling language ML-Rules as a DSL. We have shown the potential of DSL techniques for this task and also demonstrated that there are different approaches to DSL development with specific trade-offs. In the next chapter, we investigate how DSL techniques can also be useful for simulation experimentation.

7. DSLs for Specifying Simulation Experiments

In the previous chapters, we presented DSLs for modeling, which map a textual description to a mathematical object, for example a CTMC. This allows applying existing methods for working with CTMCs, such as simulation algorithms, to the modeling language. Thus, we have a way of obtaining a simulation run from a model. In this chapter, we see how DSL techniques can also be useful for wrapping complex simulation experiments around single simulation runs. The challenges here are different, however. Most importantly, there is no equivalent to CTMCs in the world of simulation experiments. Instead of abstract concepts like states and state transitions, DSLs for simulation experiments are concerned with technical issues like executing several simulation runs in parallel or combining different software artifacts. Therefore, the DSLs in this chapter are interesting from a more pragmatic perspective, supporting the conduction of simulation-based research. We start by reviewing some concepts central to simulation experiments as well as existing solutions to support simulation experiments with DSLs. Then, we present SESSL, a Scala-based object-oriented DSL for simulation experiments, and describe in what ways it facilitates experimentation. Based on SESSL, we then present a Scala-based DSL that uses purely functional programming techniques to express deterministic parallel experiments. We close with some discussion on the role of DSLs for simulation experiments in the context of simulation-based research.

7.1. Simulation Experiments

The modeling and simulation community has developed several life cycle models that structure the individual steps of simulation projects. This ranges from iterative, waterfall-style models [13] to flexible artifact-based workflows [199]. Here, simulation experiments are often used to *validate* the (executable) simulation model, making sure that the model is sufficiently accurate for its intended purpose [203]. There is a plethora of experimental validation techniques [139]. After a model is validated, further simulation experiments are conducted to explore the model behavior and answer questions about the modeled system.

In this section we review the domain of simulation experiments and the main requirements imposed on software for supporting simulation experiments. On the one hand, the term experiment implies a certain degree of scientific rigor. On the other hand, obtaining reliable results from stochastic simulation models might require non-trivial experimental methods. This makes it challenging for software to offer comprehensive support for simulation experiments.

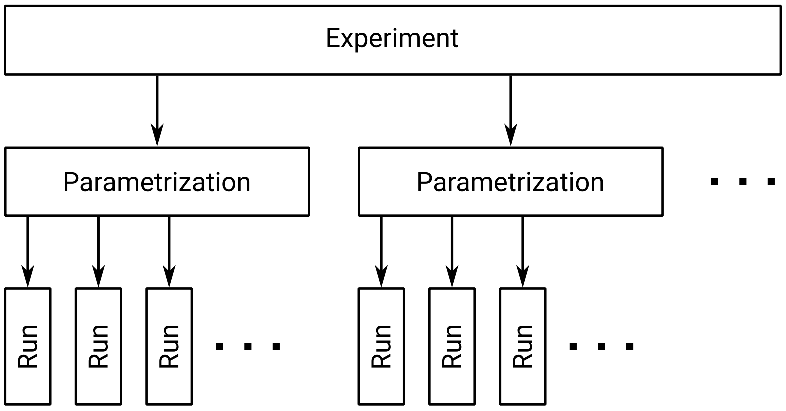


Figure 7.1.: The three layers of simulation experiment execution. Figure taken from Warnke and Uhrmacher [243].

7.1.1. Layered experimentation

Depending on the type of experiment, a variety of steps and methods can be involved, in particular for stochastic models[1]. One way to describe the structure of simulation experiments is to distinguish three hierarchical layers (see Figure 7.1) [201]. This facilitates abstracting over concrete simulation paradigms and formalisms.

Run The layers are based on the assumption that a single simulation run is a black box that deterministically produces some output for some inputs. The inputs include parameters of the model (e.g., rate constants for models of biochemical reaction networks), parameters of the simulation algorithm (e.g., which SSA variant to choose), and a source of randomness for the simulator (e.g., a seed or a stream of random numbers).

Parametrization Usually, several runs for one parametrization need to be executed to take the stochasticity into account. More precisely, these runs (*replications*) are executed with the source of randomness changed while the other parameters are unchanged. This way, point estimates and confidence intervals for single outputs can be computed [133, pp. 485ff]. Replication conditions can be used to dynamically determine how many replications to execute instead of using a predefined number.

Experiment The next level of experimentation arises from the need to understand how the simulation outputs behave for different parametrizations. Systematically exploring the parameter space is the goal of *design of experiments* methods [119, 202]. Typical applications are *sensitivity analysis* to quantify how influential different parameters or parameter combinations are, *optimization* to find a parametrization that maximizes

[1]In his textbook, Law says that "a simulation is a computer-based statistical sampling experiment" [133, p. 485]. Consequently, a lot of research into statistics has been applied to simulation and, in turn, simulation has motivated methodological development in statistics.

or minimizes some target function defined on the outputs, or a *parameter sweep*, for example for visualizing a response surface.

Decomposing simulation experiments this way allows some separation of concerns. For example, the specification of stop conditions for single simulation runs is independent from choosing which parameter combinations to explore. Other, more complex experiment types, however, may use a less strict separation of these layers [30].

7.1.2. The scientific method and reproducibility

Systematic experimentation is a core component of the scientific method [187]. Experiments are used to confirm, refute, or refine hypotheses as well as give data for formulating new hypotheses. Thus, experimentation plays a major role in many fields of scientific research including research that relies on computation.

A central property of experiments in general is reproducibility, that is the possibility to confirm the results of an experiment by repeating it (roughly speaking; a more precise definition follows below). Reproducibility is a major factor for the trustworthiness of experimental results. Nevertheless, news about failing to reproduce results appear constantly (e.g., [12, 163]). Many scientific disciplines and publications outlets feel the need to increase the reproducibility of experimental results, including results from computational experiments. For example, the ACM has initiated a Result and Artifact Review and Badging program [29]. Here, the participating journals and conferences introduce a special review stage that checks to what degree the results reported in a publication can be reproduced. The published paper then receives according badges.

While the reproducibility of real-world experiments is often affected by inputs that are hard or impossible to control, it should be much easier to make experiments conducted as computer programs reproducible. However, as the ACM initiative demonstrates, computational sciences are affected by irreproducibility as well. There has been extensive research on why that is and how to improve the situation [110, 205]. The situation for simulation experiments is similar. At the Winter Simulation Conference 2016, a panel discussion on “Reproducible research in discrete event simulation – A must or rather a maybe?” was held [231], and the Winter Simulation Conference 2018 offered a track on “Simulation Standards and Reproducibility”.

We have used reproducibility as an umbrella term for confirmable experimental results so far. However, there exist more fine-grained terms that describe different levels of this quality [110, 54, 191].

Repeatability The researcher can repeat the computation using the same software and hardware and obtain the same results. In a strict sense, this requires deterministic experiments. A less rigorous interpretation of “same results” could mean that the results follow the same distribution.

Replicability An independent researcher can repeat the original computation using the same software and obtain the same results. In some cases (e.g., performance experiments) it might also be necessary to use the same hardware to replicate the original results. In other cases the result might be largely independent from the hardware[2].

Reproducibility The original results can be produced with new, independently developed artifacts. This can include porting an algorithm to another programming language, translating a model from one modeling language to another, or running an experiment on a different kind of hardware.

In particular in modeling and simulation, more terms that are related to reproducibility can be found. For example, *cross-validation* refers to different models independently producing similar results and, thus, supporting each others credibility, which fits the above definition of reproducibility. Reproducibility is also facilitated by *reusable* artifacts, for example when reusing experiments for several related models to compare their results [184, 183]. Proper separation between model and simulator allows reusing the simulator for other models as well as running the model with another simulator. Based on these concepts, individual simulation experiments, models, and other artifacts can be related to each other. Decomposing a simulation study this way allows determining the *provenance* of the study's results [197].

7.1.3. Software support for simulation experiments

Software support for simulation experimentation has received less attention than software support for simulation modeling. Many modeling tools include only simple experimentation functions. For example, NetLogo includes the BehaviorSpace module, which allows running simple parameter sweeps [252]. However, as these tools for experimentation are part of a specific simulation system, it is not possible to treat experiments as standalone artifacts. This hampers repeating the same experiment with a model developed in a different simulation system and, thus, reproducibility.

A more versatile approach is taken by the Simulation Experiment Description Markup language (SED-ML) in the systems biology domain [237]. Based on the reporting guideline for "Minimal Information About a Simulation Experiment" (MIASE), SED-ML defines an XML-based format for specifying simulation experiments [236]. A file representing an experiment can then be imported or exported by SED-ML-compatible software. This allows distributing (and replicating as well as reproducing) simulation experiments between simulation software and between researchers. By design, SED-ML files are intended to be read or written by software rather than humans. An experiment file in itself is not executable, but is interpreted by another program. Consequently, SED-ML requires adding import/export capabilities to the actual simulation system.

A third approach to encoding simulation experiments are scripts in languages like Python or simply as shell scripts. The advantage of this approach is that the full expressive power of the underlying language is available. Scripts can compose invocations to individual software artifacts and, thus, facilitate the implementation of arbitrarily

[2]In modeling and simulation, this is related to the unambiguity of modeling formalism. If a simulation algorithm correctly implements the semantics of a modeling language, it must produce the correct results on any machine. In practice, technical issues often interfere with such assertions [111].

complex experiments. Interspersed with comments, such scripts can also transport the underlying idea of the experiment. This can be seen as a variation of Knuth's literate programming, sometime called literate computing [157]. A particular variant of scripting that support communication of experiments are interactive computational notebooks. However, as a recent survey finds, computational notebooks have weaknesses in several areas [47]. For example, missing information about the environment setup (such as installed software packages or customizations) complicates replicating results or reusing code snippets.

7.2. SESSL

Based on the previous sections, we infer three important aspects for software supporting simulation experiments:

Replicability The software should enable replicating the results of simulation experiments. The effort for installing and setting up all software to replicate an experiment should be minimized.

Flexibility To allow working with models effectively, the software must allow for a wide range of simulation experiments. This means allowing the implementation of many experimental methods or interfacing with existing implementations of experimental methods.

Reuse and Reproducibility Simulation experiments should be reusable across concrete simulation formalisms and systems to reproduce their results. Ideally, it is also possible to reuse aspects of simulation experiments.

These requirements have motivated the development of a "Simulation Experiment Specification on a Scala Layer" (SESSL). SESSL features a library that abstracts over simulation systems, offering a unified interface for specifying simulation experiments (reproducibility). Based on this interface, simulation system-agnostic features can be implemented, possibly by invoking third-party software libraries (flexibility). The interface can also be accessed via a readable and succinct DSL for experiment specification, and snippets from that DSL can be reused across experiments (reuse). Finally, SESSL experiments are directly executable and explicitly declare their dependencies, which can be obtained automatically. This allows distributing SESSL experiments which can be executed with a single click (replicability).

The name SESSL refers to the DSL as well as to the underlying implementation and library. SESSL has been initially developed by Roland Ewald between 2012 and 2014 [70], after which I took over and extended SESSL further. In the following, we introduce the fundamental ideas of SESSL and describe the changes after 2014.

7.2.1. Overview of general concepts

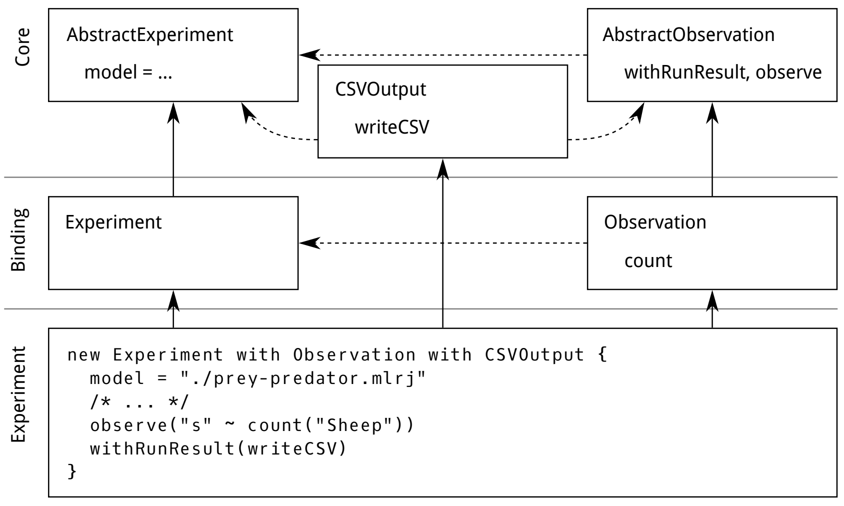
SESSL is an internal DSL that is mainly based on object-oriented principles. In particular, SESSL relies heavily on the *cake pattern* [172] for composing complex simulation

```
 1 import sessl._
 2 import sessl.mlrules._
 3
 4 execute {
 5   new Experiment with Observation with CSVOutput {
 6     model = "./prey-predator.mlrj"
 7     simulator = SimpleSimulator()
 8     stopTime = 100
 9     replications = 5
10     scan("wolfGrowth" <~ (0.0001, 0.0002))
11     observe("s" ~ count("Sheep"))
12     observeAt(range(0, 1, 100))
13     withRunResult(writeCSV)
14   }
15 }
```

Figure 7.2.: This experiment specification executes 5 replications each of 2 parametrizations of a prey-predator model defined in ML-Rules. Lines 1 and 2 import the SESSL core and the SESSL binding to ML-Rules. In line 5 the experiment object is constructed with two mixed in traits for observing outputs and writing CSV files. The following lines specify the path to the model file, the simulation algorithm to use, and the stopping and replication conditions. Two values for the model parameter `wolfGrowth` are used, and the number of `Sheep` entities is observed at the time points $0, 1, \ldots, 100$. The resulting data is written to CSV files.

experiments from individual ingredients. These ingredients are captured in *traits*, Scala's version of abstract interfaces, which can contain concrete as well as abstract members. By *mixing in* traits when creating an experiment object, the features contained in the trait become available in the experiment. This can lead to automatic adaptation of defaults. For example, to make an experiment execute simulation runs in parallel, just mixing in the corresponding trait can suffice. In most cases, however, mixed in traits enable additional configuration options for the experiment. For example, an observation trait usually allows to specify which model outputs to record during the simulation run.

To create a concrete experiment, a simulation system to execute the experiment on must be chosen. The experiment is then a Scala object of the class `Experiment`, potentially with some mixed in traits. The class `Experiment` is provided by a *binding*, which is the collection of the code specific to the selected simulation system. Apart from the experiment base class, a binding typically also contains some simulation system-specific traits. Simulation system-agnostic code, on the other hand, is assembled in the SESSL core. Class hierarchies and self-types for traits constrain how the different pieces of code can be composed and make sure that invalid combinations are rejected by the Scala compiler. The Figures 7.2



```
new Experiment with Observation with CSVOutput {
  model = "./prey-predator.mlrj"
  /* ... */
  observe("s" ~ count("Sheep"))
  withRunResult(writeCSV)
}
```

Figure 7.3.: This diagram shows some classes and traits that are used in the example in Figure 7.2 as well as some of the methods they contain. Solid arrows show sub-type relations ("is a sub-type of"), dashed arrows show self-type relations ("can only be mixed in an instance of"). Thus, for example, the trait `CSVOutput` can only be used in experiments that are subtypes of `AbstractObservation` and `AbstractExperiment`, such as the example experiment. These dependencies are checked by the Scala compiler.

and 7.3 exemplify how a simple experiment is defined in SESSL as well as how it is composed of classes and traits.

Technically, the individual configuration options for an experiment object are set in the constructor block of an anonymous class. Which options are available depends on the experiment class and the mixed in traits. Many of the settings employ a declarative syntax with statements of the form `setting = value`. Such statements are transformed to `setting_=(value)`, which is just a method call on the newly created object. This is an example for how Scala supports defining internal DSLs. SESSL exploits techniques like this to increase the readability of the language.

Once an experiment object has been created, it can be run by calling the function `execute` (which is part of the SESSL core) with the experiment as an argument[3]. This function call can be put in the main method of a Scala object, a Scala script, or any other entry point for execution. By being directly executable, SESSL experiments do not depend on third-party tools for interpretation or execution. It is also possible to wrap the execution of an experiment in a function and, thus, abstract over the experiment in further Scala code. This approach is used by SESSL to implement, for example, simulation-based optimization, where an experiment is executed as part of the target function of a simulation algorithm (see Section 7.2.2). Users can also write arbitrary code

[3]With only one experiment as the argument, Scala allows using curly braces instead of parentheses to give `execute` the look and feel of a control structure rather than a function (see Figure 7.2).

"inside" experiments to add features to the anonymous class they define. Event handlers are one feature that exploits this to run user-defined functions (callbacks) on the results of every simulation run, for example. It is also possible to combine SESSL experiments with third-party software this way.

7.2.2. Extensions and refinements

User feedback and technical developments motivated some extensions and refinements based on the fundamental ideas of SESSL as described in the previous section. One significant set of additions are new bindings for simulation systems, including ML-Rules 2 [238], ML3 [244], and pSSAlib [174]. Moreover, we added implementations for some more advanced experimental methods. They exploit the fact that, as discussed in Section 7.1.1, simulation runs are often considered black boxes. This allows us to implement methods such as statistical model-checking, sensitivity analysis, and bifurcation analysis in SESSL in a simulation system-agnostic manner. In the following, we give a short overview of the implementations of those advanced experimental methods as well as other extensions in the SESSL core. Based on these extensions, we will be able to discuss the utility of SESSL as an experiment specification tool in more detail.

Statistical model-checking

Statistical model-checking allows deciding whether a random simulation run of a model satisfies a given formal property with at least a certain probability[4] [2]. In comparison to numerical model-checking, statistical model-checking does impose fewer requirements on the model under study. It treats the model as a black box and relies on sampling trajectories via simulation, instead of, for example, exhaustive search of the state space. The property to check is usually defined as a temporal logic formula to express statements about the behavior of the model over time.

SESSL includes a trait to support statistical model-checking that adds two configuration options to an experiment. First, it allows adding a property to check. Properties are essentially functions of simulation run results to a boolean value; temporal logics are one way to implement such a function. SESSL offers a simple DSL to express formulas in Signal Temporal Logic (STL) [146]. It equips the typical LTL operators eventually (F), always (G), and until (U) with intervals, enabling to restrict them temporally. The atomic formulas are predicates on the model outputs. For example, the formula $F_{[10,20]}a > b$ expresses that the model output a must be greater than the model output b in at least 10, but at most 20 time units. In SESSL, this formula can be written as:

```
F(10, 20)(OutVar(a) > OutVar(b))
```

Second, besides the property, a statistical model-checking experiment in SESSL also requires a test procedure. The test procedure takes a parameter p and decides, based on some simulation runs, if the real probability θ that a simulation run satisfies the property

[4]Less frequently, statistical model-checking is used to estimate *with which probability* a random simulation run of a model satisfies a given formal property.

is greater than p. As statistical model-checking is based on hypothesis testing, these results are probabilistic, but provide bounds for the probability for Type I and Type II errors. A Type I error occurs when the reported result is $\theta < p$ although $\theta \geq p$, and, conversely, a Type II error occurs when the reported result is $\theta \leq p$ although $\theta > p$. SESSL includes the two test procedures most frequently discussed in literature: the *single sampling plan* and the *sequential probability ratio test* [208, 137]. For each test procedure, the values for p, the maximum Type I and Type II error probabilities α and β, and further statistical parameters can be given. SESSL then uses this information to automatically infer how many runs to execute to decide the statistical model-checking question.

Optimization and Analysis

Not only a simulation run, but a whole experiment can be considered as a black box for statistical analysis. SESSL exploits the power of its host language Scala for such application scenarios. Essentially, the experiment execution is wrapped in an anonymous function that takes as arguments the inputs of the black box and uses a callback to write the outputs to a mutable container.

```
analyze { (inputs, output) =>
  execute {
    new Experiment with /* experiment traits */ {
      /* experiment setup */

      for ((input, value) <- inputs.values)
        set(input <~ value)

      withExperimentResult { results =>
        output <~ /* calculate target function from results */
      }
    }
  }
} using new AnalysisMethod {/* */}
```

Here, the first and the last line wrap a SESSL experiment in a functions with the arguments `inputs` and `output`. The `inputs` are a parametrization that is applied in the experiment. From the result of the experiment, a value is computed and written into `output`. The analysis method can then run the function wrapping the experiment as it sees fit. The concrete analysis method is specified and configured after `using`, similarly to how SESSL experiments are created.

This scheme was originally introduced in SESSL by Roland Ewald to implement simulation-based optimization, where the outer function is a meta-heuristic optimization algorithm from the Opt4j library [144]. However, it is also applicable for other analysis methods. In particular, we implemented prototypical support for sensitivity analysis and bifurcation analysis and were able to factor out the common structure into an abstract analysis method trait [39].

Typed observations

Typed observations are an example for a refinement that exploits that SESSL is an internal DSL. In the original SESSL, observing a model output was specified by relating strings to each other. For example, in the ML-Rules binding, the following line binds the observable `c` to the number of observed `Cell` entities in the model.

```
observe("c" ~ "Cell")
```

Whereas this is succinct, it loses valuable information instead of exposing them for other parts of the experiment. First, the number of `Cell` entities is observed, which contains the information that the observed values will be numbers. The user, however, has no way of expressing this knowledge. Second, the observable is bound to the string `c`, which holds no further information about the observable it refers to. Thus, further functions on `c` that depend on numeric values require an explicit type annotation at each call site:

```
val trajectory = run.trajectory[Double]("c")
```

If the observable `"c"` does not exist or does not contain observations of type `Double`, this causes a runtime error.

As an alternative, we introduced a new syntax to specify observations:

```
val c = observe(count("Cell"))
```

Here, `count` is a function that transforms a given specification of a species to an `Observable[Double]`, where `Observable[_]` is a new type constructor. The observable is registered in the experiment by the call to `observe`, which also returns it, storing it in `c`. Note that `c` is now a Scala variable with the informative type `Observable[Double]` rather than a `String`. Consequently, later uses of `c` can use this type information, and helper functions can be constrained to only operate on numeric observables. Manual type annotations are no longer necessary, and the compiler can typecheck all function calls.

```
val trajectory = run.trajectory(c) // inferred to be Trajectory[Double] by the compiler
```

Reifying observables as a data type in SESSL is a step in refining the underlying semantic model of simulation experiments in general. It allows the user to express her intent when defining the outputs of a simulation, the resulting constraints are propagated through the experiment, and correctness conditions can be checked by the compiler.

Summary

The extensions outlined above as well as the newly implemented bindings for simulation systems illustrate three ways in which SESSL contributes to developing software for supporting simulation experiments.

First, by developing bindings for several simulation systems the commonalities between them become evident. In particular, simulation experiments typically follow the three layer structure as described in Section 7.1.1. Consequently, bindings need to manage simulation runs, aggregate them to parametrizations, check replication conditions etc., which leads to duplication of code among bindings. To avoid this, we factored out the shared features into a trait called `DynamicSimulationRuns`, which abstracts over bindings or simulation systems that are able to issue new simulation runs during the experiment. This way, the individual bindings only contain the code specific to the targeted simulation system. The trait `DynamicSimulationRuns`, on the other hand, represents a generalization of such bindings.

Second, the same effect can be observed for experiment types. One example is the common shape of analysis and optimization experiments as outlined above. By abstracting over these experiment types, we were able to factor out shared control flow logic. The remaining code is specific to the experiment type and, for example, translates between SESSL and a third-party library for sensitivity analysis. The implementation of statistical model-checking has another interesting aspect. It directly depends on the trait `DynamicSimulationRuns` (via its self-type, see Section 7.2) and exploits it to inject its hypothesis testing into the evaluation of replication conditions.

Third, we were able to refine many interfaces between different components. The typed observables discussed above are a good example for this. By capturing information about the types of observed values, the corresponding code becomes much more expressive. Similarly, the trait `DynamicSimulationRuns` defines the type of the method for starting a simulation run, clearly stating that the inputs are a random seed and the map of parameters, as described in Section 7.1.1.

With the extensions described above, SESSL expresses more information about simulation experiments. With generalizations of experiments and experimental methods, we can define the communication between them more precisely on a higher level of abstraction. This way, the semantic model underlying SESSL (see Section 2.3) has become more expressive. The structure of the classes and traits in the SESSL core and its bindings can be considered a metamodel as on object-oriented DSLs. For example, a part of this structure is visualized in Figure 7.3, not unlike the UML diagrams mentioned in Section 2.3.2. In Section 7.3 we investigate how similar information can be expressed in the functional paradigm.

7.2.3. Publishing SESSL experiments

An important aspect of replicability in simulation-based science is the ability to publish and distribute executable experiments. Again, SESSL exploits that experiments are valid, executable Scala code. However, to actually run that code, it must be compiled and all necessary compile-time and run-time dependencies must be available. For replicability, it is also important that the correct versions of the dependencies are used.

SESSL relies on Apache Maven[5] to handle these technical issues. Maven is an industry-grade software management tool used primarily in the Java ecosystem. As it runs on the JVM, it can be used on all major operating systems. Many plugins are available to extend Maven and inject new features, including a plugin for compiling and running Scala code. In addition, Maven can utilize online artifact repositories in which sources, documentation, and binaries of diverse libraries and other software projects are available.

[5]maven.apache.org

We exploit this to offer a "SESSL quickstart project" with two important files. First, it contains a Scala file with an executable experiment. Second, it contains a file called `pom.xml`, which is the project-specific Maven configuration file. This file mainly consists of two elements, one specifying the immediate dependencies of the experiment and one specifying the name of the Scala file that contains the executable experiment. When started with the command line invocation `mvn scala:script` Maven automatically runs the experiment, which includes the following steps:

- The immediate dependencies and their transitive dependencies are determined and, if they are not present on the system, downloaded from online repositories. The immediate dependencies are typically the SESSL bindings used in the experiment, which in turn depend on the SESSL core and other libraries. The version for all dependencies is fixed.
- The Scala compiler and associated artifacts are downloaded if they are not present on the system. The Scala version is declared by the SESSL artifacts.
- The experiment file is compiled using the downloaded compiler and dependencies.
- The compiled experiment is executed.

Maven stores the dependencies of the experiment and the Scala compiler on the user's computer, where they are also available for future execution of the same experiment or others. This way, the files are only downloaded once. In addition, the quickstart project does not need to contain any binaries, and experiments can be distributed as small archives. All SESSL artifacts are published in the standard online repository Maven Central via Sonatype's Open Source Software Repository Hosting program[6].

To make distributing and running SESSL experiments even more convenient, we applied the Maven wrapper[7] to the quickstart project. This tool relieves users of manually installing Maven. All Maven invocations `mvn` are replaced with `mvnw`, and the first invocation downloads a Maven executable and stores it in the same way as project dependencies. In addition, we provide executable scripts for Windows and Unix that wrap the Maven wrapper call. Experiments packaged this way can be published as archives of about 50kB. Once downloaded, they can be compiled and run with a single click (assuming that all required software is JVM-based). Thus, the only requirements for replicating a SESSL experiment are an installed JVM and an internet connection.

Packaged experiments can be modified and redistributed in several ways. The experiment in the Scala file can be adapted, for example to change parameter values. Typically, the model is defined in a separate file in the package and can be adapted as well. Third, the technical experiment setup in the `pom.xml` file can be modified, for example to update the dependencies' versions and benefit from bug fixes or performance improvements. This way, SESSL experiments can be a valuable instrument to document, distribute, and reuse simulation experiments.

[6] https://central.sonatype.org/pages/ossrh-guide.html
[7] https://github.com/takari/maven-wrapper

7.3. Purely Functional Simulation Experiments

SESSL is fundamentally object-oriented, which allows composing experiment aspects by inheritance and mixing in traits. However, this software design pattern, the so-called "cake pattern" [172], has recently been criticized[8] [85]. One problem is that the dependencies between the different traits are, in some sense, left implicit. For example, a trait can access members of its self-type, but those members are only defined in the self-type. Thus, the individual traits do not define clear interfaces between them, which can complicate adding new features in traits or in-place in experiments.

Second, with increasingly complex experiments it is increasingly challenging to keep SESSL experiments deterministic and, thus, strictly repeatable. As discussed in Section 7.1.1, each simulation run must be initialized with a random seed. For simple experiments, such as a parameter scan with a fixed number of replications per parametrization, the number of needed random seeds can be determined before running the experiment. This makes it possible to generate the seeds in advance. In more complex experiments, however, for example when using dynamic replication conditions, the number of needed random seeds depends on the immediate results of the experiment. If, in addition, simulation runs from multiple parametrizations are executed in parallel, race conditions threaten the determinism of assigning seeds to the simulation runs.

Whereas it is possible to solve this problem in imperative/object-oriented approaches such as SESSL, the functional paradigm provides determinism "for free". Therefore, we study how deterministic complex simulation experiments can be implemented via idiomatic functional programming in this section. The basic idea is to express a simulation experiment as a pure function, that is a function that is deterministic and has no side effects [171, p. 222]. In the remainder of this section, we show how established functional programming concepts and libraries can be utilized to implement deterministic, parallelizable, extendable simulation experiments. We use SESSL as a point of reference regarding the features and also the surface syntax of this new approach. The code snippets are kept to a minimum; a more technical description can be found in Appendix B.4.

7.3.1. Simple experiments

As a starting point, we consider a single simulation run. In Section 7.1.1 we discussed that such a single simulation run needs at least two inputs: a random seed and parameters for the model and the simulation algorithm. Given the same seed and the same parameters, repeating a simulation run should yield the same results. Thus, a simulation run can be represented by a pure function:

```
def sim(params: Params, seed: Seed): Result
```

given some types `Params` (typically a map of parameters to values), `Seed` (e.g., a number), and `Result` (e.g., a map of observables to trajectories). Similarly as SESSL's bindings, the function `sim` generalizes simulation runs. For example, this function could additionally

[8] Also see the discussion on the Scala mailing list: http://www.scala-archive.org/The-cake-s-problem-dotty-design-and-the-approach-to-modularity-td4640697.html

take a model as its input, a stop condition, or information about what outputs of the simulation run to observe. To integrate these aspects, a higher-order function taking these additional arguments can be written that returns the function `sim` with two arguments as shown above. This way, the functional paradigm helps to separate settings for the complete experiment from settings for a single simulation run. As a result, we can use the function `sim` as an abstraction over different simulation systems that require different experiment-wide settings. The abstract types `Params`, `Seed`, and `Result` further illustrate the independence of a specific simulation system.

We continue constructing the next layer from Section 7.1.1. For one parametrization, that is one value for the argument `params`, the simulation run shall be started with several random `seeds`. The idiomatic way to express a computation that requires several random values is the state monad (see Appendix B.4.1 for details). Libraries for functional programming such as cats[9] contain code that implements the state monad as well as a number of associated functions. Using these, we can express one parametrization with parameters `params` and `n` replicated runs as follows:

```
def replications(params: Params, n: Int) =
  rndSeed.map(sim(params, _)).replicateA(n)
```

where `rndSeed` draws a random seed in the state monad. The idiomatic combinator `replicateA` repeats seeding and running `sim` `n` times.

To express a parameter scan, parametrizations need to be further composed by varying the argument `params` of `replications`. The starting point is a list of parameter combinations `paramss` to explore.

```
def scan(paramss: List[Params], n: Int) =
  paramss.traverse(p => replications(p, n).map(p -> _)).map(_.toMap)
```

Here `traverse` expresses the embedding of the computation in the state monad. The two calls to `map` structure the output data as a map of parametrization to a list of results for that parametrization.

We now have covered many of the SESSL features shown in Figure 7.2. If necessary, experiment-wide configurations such as selecting the model, the stop condition, and configuring the observation can be realized by a higher-order function that produces the function `sim`, which is then used throughout the experiment. We have seen how the number of replications and the list of parametrizations are used to invoke `sim` with varying arguments. The implementation is considerably simpler than in SESSL. Yet, two minor aspects of the experiment in Figure 7.2 are still missing.

First, SESSL provides features to succinctly specify parametrizations, for example as a full factorial design over several input parameters of the model. In our functional approach, we can achieve the same with Scala's built-in for-comprehensions. For example, the following snippet produces a list of maps, where each map assigns a value to the parameters `"x"` and `"y"`:

```
for {
```

[9] https://typelevel.org/cats/

```
  x <- 1 to 5
  y <- 10 to 100 by 10
} yield Map("x" -> x, "y" -> y) // Map(x -> 1, y -> 10), Map(x -> 1, y -> 20), ...
```

This results in 50 maps, which can be used as the argument `paramss` in the function `scan` above. Similarly, other experimental designs can be created with plain Scala code.

Second, the SESSL listing in Figure 7.2 writes the results of each run to a CSV file. In SESSL this is implemented with an event handler that is invoked on the runs' results. In contrast, the functional paradigm allows expressing this via function composition. For example, `sim` can be wrapped in a function that writes the results before returning:

```
def write(result: Result): Result = { writeCSVFile(result); result }
def simAndWrite(params: Params, seed: Seed) = write(sim(params, seed))
```

Note that these are not pure functions anymore, as writing a file is a side effect. An alternative, more idiomatic approach is to run the entire experiment as a pure function and then write the results of the entire experiment to the disk.

We have seen that many aspects of running simulation experiments can be covered succinctly with idiomatic functional programming. However, we have only considered very simple experiments that execute a fixed number of runs sequentially, which is also not hard to implement deterministically in an imperative program. Next, we extend the approach to more complex experiments, where the functional paradigm leads to significantly simpler code than the imperative approach in SESSL.

7.3.2. Parallelism and replication conditions

Parallelizing computations is a natural strength of functional programming [96]. The idiomatic approach is similar to the code we showed above, where we embedded a sequential experiment in the state monad to supply random numbers as seeds for each run. But now, we additionally wrap computations in an effect monad, often called `IO`, which allows to represent computations as values and composing them. There are two ways of adding the `IO` monad to the state monad, both of which represent different kinds of control flow. Being able to capture this difference in types is one example for the utility of the functional paradigm for parallelization. As before, we abstract from technical details as much as possible and refer the interested reader to the more in-depth description in Appendix B.4.3

The first variant uses an applicative functor, which expresses computations with a "fixed structure" [151]. In our case, this fixed structure refers to the fixed number of simulation runs, which means that the number of needed random seeds is fixed as well. Thus, the needed seeds can be generated before starting any simulation run. The seeded simulation runs are then executed in parallel by embedding them in the `IO` monad. This way, the sequential experiments with a fixed number of runs as presented in the previous section can be parallelized.

The second variant uses a monad composed from the `IO` monad and the state monad to express computations in which "the value returned by one [sub]computation" is able "to influence the choice of another" [151]. This is realized by giving the monadic computation the ownership of the source of random numbers for its whole duration, enabling it to generate an arbitrary number of random seeds and, therefore, start an arbitrary number of simulation runs. Note the difference to the first variant, where random numbers are only generated in the beginning of the computation. As a consequence, it is not possible to parallelize such monadic computations[10].

We can express experiments that combine the advantages of both variants by composing them hierarchically. Essentially, the overall experiment is a monadic computation, but its subcomputations are parallel batches embedded in the applicative functor. This way, we can express the same control flow that the implementation in SESSL uses. For example, the simulation runs in an experiment with dynamic replication conditions are executed in parallelized batches, and the condition is evaluated after all runs of a batch are completed.

7.3.3. Complex simulation experiments

We have covered functions that are concerned with composing overall deterministic experiments from single simulation runs. To express more complex experiments in the functional paradigm, we now interleave simulation runs with other functions. We discuss two examples of complex simulation experiments that are implemented in SESSL: statistical model-checking and simulation-based optimization.

Statistical model-checking

To implement statistical model-checking, we need to add three elements to the experiments as discussed above. First, we need to define a property to check, for example in temporal logic. Second, we need to actually check for each simulation run's result whether it satisfies the property. Third, we need to integrate test procedures that control the number of simulation runs to execute. Particularly the first two elements can be expressed very well in the functional paradigm.

The property to check is essentially a function `Result => Boolean`. There are several ways to define such a function, potentially depending on the concrete `Result` type. One way is to directly define a predicate on the simulation results, for example to check whether some observable has crossed a threshold:

```
def prop(r: Result): Boolean = values(r, x).exists(_ > 1000)
```

where `values(r, x)` yields the observed values for an observable `x` in the simulation results `r`. A more general way to define a property is to write a curried function that takes a formula in some temporal logic (e.g., STL [146]) in addition to the results.

```
def stl(f:STLFormula)(r: Result): Boolean = /* check whether r satisfies f */
```

for some type `STLFormula`, which represents STL formulas as a mini-DSL similar as in SESSL (also see Barringer and Havelund [16]). Then, for some concrete simulation experiment, the expression `stl(f)` with `f` being an STL formula yields a function of type `Result => Boolean`, which is exactly the type of a property.

[10]One way to circumvent this is to split the source of randomness to be able to give one to each monadic computation. However, splitting random number streams might introduce statistical bias [204].

As a property is represented by a function, evaluating a property on a run's results (the second element above) is just function composition. For example, `prop(sim(p, s))` determines whether the simulation run with parameters `p` and seed `s` satisfies the property `prop`.

The third element, controlling the number of replications, can be realized similarly as in SESSL: by reusing the existing measures for setting replication numbers and replication conditions. As discussed in the previous section, replications conditions are evaluated on the results obtained so far. This allows implementing the statistical model-checking test procedures such as the sequential probability ratio test (see Section 7.2.2).

In summary, statistical model-checking experiments can be expressed very well in the functional paradigm. The properties to check can be represented and passed around as plain functions, and property languages such as temporal logics can be integrated via currying.

Simulation-based optimization

As discussed in Section 7.2.2, optimization experiments in SESSL are already expressed with functional concepts. In particular, the target function for the optimization algorithm is implemented by wrapping the execution of a SESSL experiment in a function. This target function can then be invoked at will by the optimization algorithm for different parametrizations.

The same approach can be expressed directly in a purely functional approach. In SESSL, an event handler is used to write the value of the target function to a mutable container when the experiment finishes. This way, the results of all runs are available to determine the target function's value. In other word, the event handler implicitly expresses a function `List[Result] => TValue`, where `TValue` is the result type of the target function. In the functional paradigm, we can explicitly define such a function. Similarly as for statistical model-checking, this function can then be composed with a function that produces the results for a specific parametrization to obtain the complete target function.

The second issue is the determinism of the optimization experiment. In general, optimization algorithms themselves require a source of randomness, for example to randomize the mutations in a genetic algorithm. Thus, the optimization algorithm randomly chooses parametrizations to execute, and (when using dynamic replication conditions) this influences how many random seeds for simulation runs are required, which in turn influence the next choice of the optimization algorithm and so on. To keep the whole process deterministic, we pass the source of randomness between the optimization algorithm and the simulation runs. This sequence of computations can be expressed in the monadic approach discussed above.

In comparison to SESSL, optimization experiments benefit from the functional paradigm by expressing the composition of the outer function (i.e., the optimization algorithm) and the inner function (i.e., the target function based on simulation runs) directly. This allows a simpler formulation of the target function and facilitates making the overall experiment deterministic.

7.3.4. Expressing experiments in a DSL

As we have seen above, many aspects of simulation experiments are directly expressible in the functional paradigm. By using appropriate libraries, the control flow in parallel, complex experiments can often be captured by composing a few idiomatic combinators. To complete the presentation of this approach, we now illustrate how the implementation can be wrapped in an internal DSL. As before, we relate this new DSL to SESSL.

The following listing shows a statistical model-checking experiment in SESSL. It uses a binding that runs simulations of Law's inventory model[11] [133, sec. 1.5] and checks whether the output variable `AverageTotalCost` is less than 150. The model is executed with a specific parametrization and parallel runs are executed in batches of 10. Finally, the experiment specification includes the statistical parameters for the sequential probability ratio test.

```
new Experiment with Observation with ParallelExecution with StatisticalModelChecking {
  prop = (_, outputs) => outputs(AverageTotalCost).values.last.asInstanceOf[Double] < 150
  set("s" <~ 20, "S" <~ 60)
  batchSize = 10
  test = SequentialProbabilityRatioTest(
    p = 0.9,
    alpha = 0.05,
    beta = 0.05,
    delta = 0.05
  )
}
```

The next listing shows the same experiment in the functional approach. Similarly as in the SESSL experiment, the individual settings are specified in a declarative style with `setting = value`. In contrast to SESSL, where such lines are object-oriented setters, the functional approach uses named function arguments.

```
sequentialProbabilityRatioTest(
  prop = result => result(AverageTotalCost) < 150,
  batch = replications(
    params = Map("s" -> 20, "S" -> 60),
    n = 10
  ),
  p = 0.9,
  alpha = 0.05,
  beta = 0.05,
  delta = 0.05
)
```

As the above example shows, the surface syntax of SESSL and the new approach is quite similar. One notable difference is that the property specification in SESSL is

[11]The specifics of the model are not relevant here.

much more verbose than in the functional approach. However, this is not caused by SESSL's design principles, but rather an implementation artifact. More importantly, the way experiments are composed is different. In SESSL, an experiment receives mixed in traits upon creation, such as `ParallelExecution` and `StatisticalModelChecking` above. The resulting experiment objects combines the features of all these traits. In contrast, the functional approach composes functions to define the overall experiment. For example, the statistical model-checking experiment is defined by composing the hypothesis test (`sequentialProbabilityRatioTest`) with the function `replications`, which executes a batch of simulation runs in parallel.

7.3.5. Summary

We presented a purely functional DSL for experiment specification that builds upon SESSL in several ways. In particular, we adopted the idea of abstracting over different simulation systems. Moreover, the replicability techniques for packaging and distributing experiments detailed in Section 7.2.3 can be adopted without changes in this new language. Complex experiments are composed from simpler components, although the paradigm of composition differs: SESSL uses object-oriented concepts, whereas the new approach relies on functional composition. This difference leads to three advantages. First, in the new approach it is simpler to specify deterministic experiments, as experiments are defined as pure functions. Second, the new approach makes it easier to abstract over experiments by extracting function arguments. For example, the above statistical model-checking experiment could be modified to allow factoring out the function `sim` which executes the actual simulation runs. This would facilitate reusing the experiment specification for different models or even different simulation systems. Third, the new approach is more flexible than SESSL. Whereas in SESSL the user was able to inject custom code in event handlers, the functional experiments can be augmented through function composition at any stage.

7.4. Adapting and generating experiments

We presented two different Scala-based DSLs for defining executable simulation experiments, employing the object-oriented and functional paradigm, respectively. Defining experiments in a DSL facilitates considering them as standalone artifacts. This can be exploited, for example, when capturing simulation studies in artifact-based workflows [199]. Here, experiments have their own life cycle, formalizing the modification and reuse of experiments.

One idea is to take the simulation experiments associated with a model and reuse them when the model is extended [184]. Similarly, when composing several models the experiments associated with the individual models can be reapplied to the composed model [183]. Another approach is to generate new experiments based on model documentation [200]. All these operations on experiments can be done semi-automatically by providing according algorithms. The implementation of such algorithms relies on an explicit representation of simulation experiments, for example with a DSL. In the following, we discuss how DSL techniques can support adapting and generating experiments.

To design algorithms that operate on experiments, a suitable representation of experiments is required. In the DSL paradigm, there are two main ways to do that. First, experiments can be processed in their concrete syntax. Second, an abstract syntax representation can be employed. We illustrate both ways using SESSL.

When operating on the concrete syntax of textual languages, algorithms work on source code. For example, to adapt a SESSL experiment and make it use a different model file, it suffices to find a line like

```
model = "oldModelFileName"
```

and replace it with

```
model = "newModelFileName"
```

The modified experiment can be saved in a new file and be distributed like any other SESSL experiment. It is also executable. We employed operations on the concrete syntax, for example, to generate experiments based on templates [200].

The downside of this approach is that such textual modifications essentially rely on an informally specified concrete syntax. In the example above, the specification of the model file must not necessarily stand in one line or on its own line. As `model = ...` is just a method call, it can also be factored out into another function or supertype of the experiment and not even be in the same file. The underlying problem is that such a line in SESSL is not a declarative key-value-assignment (although it has that look-and-feel), but a mutating method call on an object.

To better capture the true nature of experiment adaptations such as changing the model path in SESSL, it is possible to work on the abstract syntax level. The operations are not defined on source code, but on objects. For example, to adapt the model of an existing experiment object `exp` it suffices to call a method on that object:

```
exp.model = "newModelFileName"
```

We employed this approach[12], for example, to modify experiments in a visual analytics setting [246].

This approach has two disadvantages. First, it is not possible to obtain a textual description (i.e., source code) for an experiment after it has been adapted. The reason for that is also the second disadvantage. Not all aspects of an experiment are reified as fields of objects. SESSL allows and even encourages injecting Scala code for several experiment aspects. Whereas the model file is just a field with a string, other aspects like event handlers or result postprocessing are typically expressed with plain Scala code. In other words, SESSL combines deep embedding and an explicit abstract syntax with a shallow embedding of plain Scala code. The former is open for analysis, adaptation, and conversion to text, but the latter is not.

[12]Note that this in this case, the experiment object is mutated. However, immutable objects could be handled similarly with a copy-on-write approach, for example with the `copy` method on Scala's case classes: `val newExp = exp.copy(model = "newModelFileName")`

Similarly as for modeling languages, the trade-offs of deep and shallow embedding become evident. Shallow embeddings allow seamless integration of the DSL with its host language, but the resulting experiment can only be executed, not inspected or modified. This is also the approach taken by the functional implementation presented in Section 7.3. As illustrated above, to process (e.g., modify) shallowly embedded experiment descriptions in the course of a simulation study the source code (concrete syntax) representation is often the only choice. Deep embeddings, on the other hand, facilitate more fine-grained management and adaptation of experiments in a simulation study. An abstract syntax gives a rigid frame for generating, analyzing, or modifying experiments, and the danger of introducing errors can be minimized. But, as exemplified by SESSL, it is beneficial for flexibility to let the user deviate from the abstract syntax and inject host language code in some points. An example for a language strictly following the deep embedding approach is SED-ML, which uses an XML Schema to specify the abstract syntax of simulation experiments [237].

One way forward could be to formalize the distinction between deeply and shallowly embedded language constructs as proposed by Svenningsson and Axelsson [222]. The decisive step here is to define interfaces in such a way that the shallowly embedded parts can be evaluated to equivalent deeply embedded ones. In SESSL, for example, this would mean to design tailored interfaces for describing event handling and other elements that require more flexibility. Vice versa, the core experiment structure in the functional approach could be reified in a deep embedding and according interfaces be offered for extending the structure.

7.5. Summary

In contrast to simulation models, the scope of simulation experiments is much less constrained. Simulation models can be mapped to CTMCs or similar abstractions, whereas simulation experiments are so diverse that their greatest common divisor is arguably “executable program”. However, due to their role in the scientific method and in statistical analysis of simulation models, there are some recurring elements of experiment implementations. Here, using a DSL can help to factor out reusable components. This can be achieved in an object-oriented or functional programming fashion.

We have seen different ways in which the appropriate design and implementation of DSLs can support important properties of simulation experiments. First, the repeatability of experiments can be facilitated by expressing them in a purely functional way, which guarantees their determinism. Second, the replicability of experiments benefits from being able to publish them as executable packages with minimal installation and setup requirements. Third, representing experiments as standalone artifacts allows reusing them for different models to reproduce their results.

In addition, with experiments becoming more complex, statically analyzing them, generating them, and reusing them in and across simulation studies is increasingly important. Thus, supporting such operations is becoming a requirement for DSLs for experiments in addition to “just” allowing execution. For example, analysis of simulation experiments might reveal prospective provenance information in the future [152].

Designing an experiment language that allows static analysis, is flexible, and facilitates repeatability as well as reusability and reproducibility is future work (see Section 8.3). However, this chapter presented some possible ingredients.

8. Conclusion

To conclude this thesis we now summarize the key insights and discuss some directions for future work.

8.1. Summary

Based on an intuition about the rule-based modeling of biochemical reaction networks, we developed the abstract syntax and formal semantics of ML-Rules. With attributed and dynamically nested entities, ML-Rules is a very expressive language. Nevertheless, we showed that its formal semantics can be expressed in terms of a CTMC, a relatively simple stochastic process model. This allows executing ML-Rules models with established CTMC simulation algorithms.

The formal semantics is mostly concerned with determining what successor states are reachable and with what exit rate based on the current state of a model. In particular, patterns on the left rule side must be matched to the model state. We presented two implementations of ML-Rules as a DSL including the required pattern matching. The first implementation is an internal DSL that employs the functional programming paradigm to express the pattern matching through monadic composition. The second implementation uses the language workbench Xtext to provide an external DSL and the associated tooling, including the generation of object-oriented Java code. We discussed the similarities and differences of both approaches. One core result is that the external DSL implementation offers more potential for efficient execution, as it facilitates in-depth analysis of the model, paving the way for selecting and configuring a simulation algorithm. We also presented an adaptation of CTMC-based modeling for agent-based modeling with an internal DSL. This third implementation highlights the potential of DSLs to form a layer of abstraction between code for model behavior and simulation code. This facilitates succinct model descriptions and also reusable simulation algorithms.

Lastly, we investigated the utility of DSLs for the specification of simulation experiments. We presented SESSL, an internal DSL that relies on object-orientation to factor out reusable code. We also present a novel, purely functional approach to experiment specification. Whereas both of these approaches address requirements that are specific to simulation experiments, in particular repeatability, replicability, and reproducibility, neither of them is suited to statically analyze experiments. However, as we illustrate with some examples, the potential to generate, analyze, and adapt simulation experiments is increasingly important in the context of simulation studies.

8.2. Key insights

We now highlight some key insights from this thesis.

8.2.1. Models vs. experiments

We have applied the ideas of DSLs to simulation models as well as simulation experiments. In both cases the DSL represents a layer of abstraction between language user and language implementation. Nevertheless, these two use cases are quite different.

In a DSL for modeling, a model is evaluated to a description of a stochastic process. This description can then be used as input for a simulation algorithm. For example, each of our two implementations of ML-Rules contain a type that represents a model, and a concrete model can be represented as a value of this type. Such a model value essentially defines a CTMC and, thus, can be executed with an SSA or analyzed in other ways. The mapping from the language to the CTMC is specified by the formal semantics. Thus, the formal semantics defines an upper limit for the formal expressiveness of ML-Rules. As every model defined in ML-Rules can be mapped to a CTMC, ML-Rules is not more expressive than CTMCs. This way, a formal semantics can naturally limit the scope of a DSL for modeling.

In contrast, DSLs for simulation experiments are not associated with relatively simple mathematical objects like CTMCs. Their application domain has no natural limits, as many different types of simulation experiments exist. A language to express the wide range of experiment types must be extremely flexible and, therefore, arguably not less expressive than a GPL. Nevertheless, there are recurring elements in simulation experiments. By wrapping these recurring elements and allowing to combine them with each other as well as with custom code, DSLs for simulation experiments can provide value. In particular, they facilitate abstracting over individual experiments. For example, SESSL allows implementing experimental methods such as statistical model-checking in a simulation system-agnostic fashion. In addition, we have shown how DSLs for simulation experiments can facilitate repeatability (e.g., by making experiments deterministic), replicability (e.g., by making experiments easy to distribute) or reproducibility (e.g., by making experiments independent of concrete simulation systems).

DSLs can also be useful in other areas of modeling and simulation. For example, SESSL includes a sub-DSL to express temporal logics formulas in STL (see Section 7.2.2). Similar implementations have been proposed before [16]. However, the syntax and semantics of (temporal) logics are typically less complex than of languages for modeling or experimentation.

8.2.2. Formal semantics of modeling languages

One of the main contributions of this thesis is the definition of the formal semantics of ML-Rules. ML-Rules was already proposed in 2011 and implemented twice since then [149]. However, a formal definition of the language was only published in 2015 (and revised for this thesis) [239]. This is different from many other simulation modeling languages, where the semantics are defined first and only proof-of-concept implementations are provided or even no implementations at all. Retrofitting the formal semantics to the language revealed gaps between the practical implementation and the formal definition. In some sense, the formal semantics made it possible to precisely reason about the language ML-Rules, its idiosyncrasies (such as employing dynamic nesting or non-linear patterns), and implementation strategies [101]. We also discuss some differences between the formal semantics and the implementations as presented in this thesis in Section 6.2.2. In addition, the formal semantics makes sure the definition is complete and does not glance over corner cases, which is possible in informal, example-driven semantics[1]. The formal definition of a language provides the vocabulary for such discussions.

A direct consequence of defining ML-Rules formally and relating practical implementations to the formal definition is that the trustworthiness of results produced with the implementation can be assessed. As simulation of CTMCs is well understood, mapping an ML-Rules model to a CTMC creates a chain of credibility from the model definition to the simulation output (leaving software bugs aside). This is an important step in addressing the question of reproducibility in simulation, as it associates the simulation results with a CTMC instead of a specific modeling language or implementation [232].

8.2.3. Explicit abstract syntax and static analyses

Whereas the formal semantics specifies the evaluation of a model or experiment, the abstract syntax specifies the form of representation. The abstract syntax can be directly reflected in the implementation, in particular in external DSLs or in internal DSLs with a deep embedding. Such an explicit representation of the abstract syntax enables static analyses in addition to the ability of running a model or experiment. We identified static analyses as an important ingredient for DSLs for models and experiments.

If a DSL for simulation modeling provides an explicit representation of the abstract syntax, for example a metamodel as presented in Section 6.4, the model can be analyzed on the abstract syntax level. This can be exploited to evaluate a model without simulation, for example by balance analysis for CTMC-based models (see Section 3.1.3). But static model analyses can also be exploited for executing simulation runs, for example for selecting and configuring simulation algorithms as discussed in Section 6.5.3. Static analyses of rule-based simulation models can also be used, for example, for debugging models [35].

As ML-Rules is very expressive, an explicit abstract syntax is particularly useful. First, the reaction rules with the nested patterns on the left side are comparably complex. Operations on these rules are made considerably easier by capturing the syntax in an expressive form, such as a metamodel. Second, the expressiveness allows constructing models that violate the assumptions of some efficient simulation algorithms. The applicability of algorithms can be determined, however, by determining the properties of a specific given model by static analysis. In the future, automatically detecting which

[1]For instance, the formal semantics defines that for a reactant $cS(\tilde{e}; pat_r) \triangleright x$ the variable x binds to all matched reactants even if $c > 1$, although this occurs rarely.

algorithms are applicable could inform the automatic selection of a simulation algorithm for a model [101].

In simulation experiments, an explicit representation of the abstract syntax is helpful when considering experiments in the wider context of simulation studies. Here, experiments are created, adapted, or combined. These operations benefit from having a well-defined structure to work on as opposed to a textual experiment representation. In addition, an explicit abstract syntax can be used by static analysis to automatically extract provenance information from experiments without executing them. More generally, our work on SESSL has shown that using (and growing) a DSL for specifying simulation experiments exposes commonalities and differences between different types of simulation experiments as well as different simulation systems. We encoded much of this information in an object-oriented system of classes and traits, which can be considered an explicit abstract syntax or metamodel. These classes and traits allow us to add features to SESSL on an abstract, tool-independent layer.

8.2.4. Rule-based syntax in external and internal DSLs

The above points about abstract syntax apply to external DSLs as well as deeply embedded internal DSLs. In contrast, internal DSLs with a shallow embedding can be seen as a layer of syntactic sugar over their host language without an intermediate, DSL-specific abstract syntax.

Internal DSLs can combine shallow and deep embedding to reuse constructs of their host language and also define new syntactic constructs. For example, the agent-based modeling language presented in Section 6.6 makes use of Java's lambda expressions to express function literals (shallow embedding), but combines such functions in a rule object (deep embedding). This exemplifies that internal DSLs have advantages when there is some syntactic and semantic overlap between the host language and parts of the application domain. For declarative, rule-based modeling language designs, however, GPLs offer little built-in syntax. Instead, idioms of the host language can be adapted to the language domain, as we illustrate with the Scala-based ML-Rules implementation in Section 6.3 where we expressed the pattern matching as a monadic sequence. However, the functional language implementations in Sections 6.3 and 7.3 add very little syntax to their host languages. These languages stretch the definition of DSLs, which is arguably a frequent occurrence for internal DSLs.

External DSLs, on the other hand, allow designing a syntax independently of any existing language. Consequently, many simulation modeling languages are implemented as external DSLs to include language concepts that are not supported by mainstream GPLs. Our Xtext-based implementation of ML-Rules employs this approach to enable reaction rules with nonlinear patterns on their left side (Section 6.4). This approach also means that elements of GPLs must be reimplemented in the DSL, for example expressions or function definitions and invocations, possibly including type inference and scoping rules. To avoid this, DSLs can reuse existing (sub)languages. For example, Xbase is a reusable expression language in the Xtext ecosystem [64].

For other paradigms than rule-based modeling, different implementation strategies can yield an appropriate concrete syntax. DEVS and agent-based modeling, for example, map well to object-oriented programming concepts, as demonstrated by the internal DSLs Python(P)DEVS [233] and ReLogo [177].

8.3. Future work

There are several promising directions for future work.

The gap between the formal semantics of ML-Rules and the pattern matching algorithm proposed in this thesis should be investigated. It would be interesting to see how the algorithm can be formalized in operational semantics. Obviously, pattern matching the reactants from left to right would play a central role. Having defined such a semantics, the relation to the order-independent semantics as defined in this thesis could be investigated in-depth. For example, the question which reactants can still be reordered without changing the result of pattern matching could be addressed.

The expressive power of non-linear patterns in rule-based languages for modeling should be studied further. Non-linear patterns can express dynamic nesting as well as the influence of location in a grid or other kinds of discrete space. Established rule-based modeling languages like BNGL or the κ-calculus make use of non-linear patterns to encode graph rewriting. This implies that non-linear patterns could be a common formalism to relate and potentially unify different rule-based modeling approaches. In addition, non-linear pattern matching might inform the development of efficient simulation algorithms for a great variety of models.

Moreover, future work should be directed at static analyses of rule-based modeling languages. The utility of analyzing a model specification has already been shown, for example for reachability analysis in the Kappa platform [35]. It would be interesting to see under what conditions the same methods can be applied to more expressive languages like ML-Rules. Similarly, an analysis of an ML-Rules model could support the selection of simulation algorithms. Combined with an algorithm selection method, this could automatically select the optimal algorithm for a given model [99]. Another example for how static analysis support efficient simulation is the DSL for agent-based modeling presented in Section 6.6. Here, static analysis can be useful to estimate the locality of changes.

Regarding languages for experiment specification, more work is necessary to address challenges like reproducibility and flexibility. For example, methods from the area of model-driven engineering might be useful for composition of experiment specification languages as well as for composition of experiments [104, 125]. The composition of languages could be used to flexibly combine features in experiment specification languages. A core language for common experiment features could be combined with sub-languages for specific simulation systems as well as for specific experiment types. The sub-languages would then correspond to SESSL's bindings, but with a more rigid foundation and, potentially, an abstract syntax that would be open to static analysis. Composing simulation experiments has been proposed before [183]. The approach could benefit from a more formal definition of a simulation experiment's abstract syntax by defining composition on that level. By defining an unparsing[2] operation for an experimentation DSL, algorithms for generating experiments could also operate on the abstract syntax. Finally, as already mentioned in Section 7.5, a proper experiment specification DSL could support the extraction of provenance information.

[2]Xtext, for example, generates an unparser (i.e., a method to create the concrete syntax from the abstract syntax) by default.

A. Multisets

As there seems to be no commonly agreed upon convention for denoting multisets and the operations on them, we define the notations used in this thesis below. The definitions are based on Syropoulos [226], Wildberger [250], and De Nicola et al. [58].

- A multiset $\mathcal{A}$ is a pair $\mathcal{A} = (A, n_{\mathcal{A}})$, where A holds the distinct elements of the set and $n_{\mathcal{A}} : A \to \mathbb{N}$ maps each element to its number of occurrences. We call $n_{\mathcal{A}}$ the *multiplicities* of $\mathcal{A}$. It is convenient to extend $n_{\mathcal{A}}$ to an arbitrary domain by setting $n_{\mathcal{A}}(x) = 0$ for all $x \notin A$[1].
- We can also denote a multiset by listing its elements in the form $\mathcal{A} = \left\{\!\!\left| a_1^{n_{\mathcal{A}}(a_1)}, a_2^{n_{\mathcal{A}}(a_2)}, \ldots \right|\!\!\right\}$ for $a_1, a_2, \ldots \in A$.
- As a third option, a multiset can be defined in a multiset comprehension of the form $\mathcal{A} = \{\!|a|P(a,b), b \in B|\!\}$, where $P \subseteq A \times B$ is a relation of A and some set B. Note that this notation yields a multiplicity $n_{\mathcal{A}}(a) > 1$ for an a which is related to several $b \in B$.
- The empty multiset is denoted as $\emptyset$.
- An element is contained in a multiset if its multiplicity is positive:
If $\mathcal{A} = (A, n_{\mathcal{A}})$ is a multiset, then $x \in \mathcal{A} \Leftrightarrow n_{\mathcal{A}}(x) > 0$

We define some relations and operations on multisets.

- Two multisets $\mathcal{A} = (A, n_{\mathcal{A}})$ and $\mathcal{B} = (B, n_{\mathcal{B}})$ are equal iff $n_{\mathcal{A}}(x) = n_{\mathcal{B}}(x)$ for all $x \in A \cup B$. Then $\mathcal{A} = \mathcal{B}$.
- The sum of two multisets $\mathcal{A} = (A, n_{\mathcal{A}})$ and $\mathcal{B} = (B, n_{\mathcal{B}})$ is a multiset $\mathcal{C} = (C, n_{\mathcal{C}})$ with $C = A \cup B$ and $n_{\mathcal{C}}(x) = n_{\mathcal{A}}(x) + n_{\mathcal{B}}(x)$ for all $x \in C$. Then $\mathcal{A} \uplus \mathcal{B} = \mathcal{C}$.
- We can define the difference of two multisets via the sum. $\mathcal{C} \ominus \mathcal{A} = \mathcal{B} \Leftrightarrow \mathcal{A} \uplus \mathcal{B} = \mathcal{C}$.
- The submultiset relation $\subseteq$ can also defined via the sum. $\mathcal{A} \subseteq \mathcal{C} \Leftrightarrow \exists \mathcal{B} : \mathcal{A} \uplus \mathcal{B} = \mathcal{C}$.

Finally, we define the sum over a multiset of reals.

- For a multiset $\mathcal{A} = (A, n_{\mathcal{A}})$ with $A \subseteq \mathbb{R}$ we define $\sum \mathcal{A} = \sum_{a \in A} n_{\mathcal{A}}(a) \cdot a$.

[1]But $n_{\mathcal{A}}(x) = 0 \Rightarrow x \notin A$ does not hold necessarily. In other words, A may contain elements that are not contained in the multiset.

B. Listings

B.1. Non-linear pattern matching algorithm in Scala

This is a Scala implementation of the pattern matching algorithm as described in Section 6.2.1.

```
type Substitution = Map[Variable, Value]

def withBinding(substitution: Substitution, variable: Variable, value: Value): Substitution =
  if (!substitution.contains(variable))
    substitution.updated(variable, value)
  else
    substitution

def mergeAssignments(substitution1: Substitution, substitution2: Substitution): Substitution =
  if (substitution1.keySet.intersect(substitution2.keySet).nonEmpty) {
    throw new IllegalStateException()
  } else {
    (substitution1.toList ++ substitution2.toList).toMap
  }

case class Pattern(entities: List[EntityPattern], rest: Option[RestSolVariable] = None)

case class EntityPattern(
    count: Int,
    name: String,
    attributes: List[AttributePattern],
    subSol: Pattern = Pattern(List.empty, None),
    variable: Option[EntityVariable] = None)

sealed trait Variable

case class RestSolVariable(name: String) extends Variable

case class EntityVariable(name: String) extends Variable

case class Entity(
    count: Int,
    name: String,
    attributes: List[Value],
    subSol: Solution = Solution(List.empty))
    extends Value

sealed trait Value

sealed trait AttributePattern

case class Constant(value: Any) extends AttributePattern with Value

case class AttributeVariable(name: String) extends AttributePattern with Variable

case class Solution(entities: List[Entity]) extends Value

def matchEntityPattern(
    entityPattern: EntityPattern,
    entity: Entity,
    substitution: Substitution
  ): List[Substitution] =
  if (entityPattern.name != entity.name || entityPattern.count > entity.count)
    List.empty
  else {
    // check if entity variable exist and has already been assigned to
    val priorMatch = for {
      ev <- entityPattern.variable
      Entity(_, name, attributes, subSol) <- substitution.get(ev)
    } yield {
      name == entity.name && attributes == entity.attributes && subSol == entity.subSol
    }

    val canStillSucceed = priorMatch.getOrElse(true)

    if (!canStillSucceed) {
      // the pattern variable has already been matched to another entity earlier
      List.empty
    } else {
      // not matched earlier or matched same entity
      val attrsPairs = entityPattern.attributes.zip(entity.attributes)
      val afterAttributes = attrsPairs.foldLeft[Option[Substitution]](Some(substitution)) {
        case (Some(matchResult), (ap, a)) => matchAttributePattern(ap, a, matchResult)
        case (None, _)                    => None
      }
      val afterSubSol = afterAttributes.toList.flatMap { mr =>
```

```
        matchPattern(entityPattern.subSol, entity.subSol, mr)
      }
      entityPattern.variable.fold(afterSubSol)(ev => afterSubSol.map(withBinding(_, ev, entity)))
    }
  }

def matchAttributePattern(
    attributePattern: AttributePattern,
    attribute: Value,
    substitution: Substitution
  ): Option[Substitution] =
  (attributePattern, attribute) match {
    case (Constant(v), a) if a == v                           => Some(substitution)
    case (v: AttributeVariable, a) if !substitution.contains(v) => Some(withBinding(substitution, v, a))
    case (v: AttributeVariable, a) if substitution(v) == a      => Some(substitution)
    case _                                                      => None
  }

def matchPattern(
    pattern: Pattern,
    solution: Solution,
    substitution: Substitution = Map.empty
  ): List[Substitution] = pattern match {
  case Pattern(Nil, None) => List(substitution)
  case Pattern(Nil, Some(rest)) =>
    if (substitution.contains(rest) && substitution(rest) != solution)
      List.empty
    else
      List(withBinding(substitution, rest, solution))
  case Pattern(ep :: tail, restSol) =>
    for {
      (e, idx) <- solution.entities.zipWithIndex
      m <- matchEntityPattern(ep, e, substitution)
      newSol = Solution(solution.entities.updated(idx, e.copy(count = e.count - ep.count)))
      result <- matchPattern(Pattern(tail, restSol), newSol, m)
    } yield result
}
```

B.2. The nondeterministic state monad in Scala

This is a Scala implementation of the nondeterministic state monad as described in Section 6.3.2.

```
case class NonDetState[S, +A](run: S => List[(S, A)]) {
  def map[B](f: A => B): NonDetState[S, B] = NonDetState { s =>
    for {
      (s1, a) <- run(s)
    } yield (s1, f(a))
  }

  def flatMap[B](f: A => NonDetState[S, B]): NonDetState[S, B] = NonDetState { s =>
    for {
      (s1, a) <- run(s)
      (s2, b) <- f(a).run(s1)
    } yield (s2, b)
  }

  def withFilter(p: A => Boolean): NonDetState[S, A] = NonDetState { s =>
    for {
      (s1, a) <- run(s)
      if p(a)
    } yield (s1, a)
  }
}
```

B.3. Code generation in Xtext

Section 6.4 describes an Xtext-based implementation of ML-Rules. This appendix gives some additional technical background. We first look into the general approach for generating code for an ML-Rules model, and then give an in-depth explanation of the code generation for the pattern matching in a reaction rule.7

B.3.1. Generating Java code for an ML-Rules model

In Xtext projects, code is typically written in Xtend, a GPL that is largely similar to (and compiled to) Java. One of its most useful features of Xtend is the support for multi-line template expressions, which allows defining readable, indented code templates [21, p. 57f.]. Inside the templates, expressions can be inserted in guillemets «», including conditional expressions, loops, or calls to other code generation methods.

Using Xtend templates, much of the actual code generation can be simplified. Most of the code to generate is fixed and only few expressions need to be used in the templates. For example, to generate the Java class `Model`, we always use a template like the following:

```
package «packagePath»;

// omitted: imports
```

```
public class Model implements IModel {
  // omitted: contents
}
```

Here, the package path is inserted in the package declaration of the class, and in the body of the class all necessary Java code can be generated.

To illustrate how the generated code looks like, we consider the following simple ML-Rules model.

```
A()[];
B(num);

aString: 'foo';
aNumber: 3 + 4 * 12 + 2;

>>INIT[A[2 B(aNumber)]];

A[B(x):b + B(x) + rest]:a -> A[rest] @ #b * (#b - 1);
```

In the following we highlight some noteworthy snippets from the Java code that is generated for this model. First, for the species `A` and `B`, standard Java classes `A` and `B` are generated:

```
public class A implements ICompartmentalSpecies {
  public final ISolution subSol;
  public A(ISolution subSol) {
    this.subSol = subSol;
  }
  // hashCode, equals, toString, copy
}

public class B implements ISpecies {
  public final double attribute_0;
  public B(double attribute_0) {
    this.attribute_0 = attribute_0;
  }
  // hashCode, equals, toString, copy
}
```

The generated code reflects for each species the declared attributes as well as whether the species is a compartment. The types of the attributes are the Java equivalents of the types declared in the ML-Rules model. In the generated `Model` file, code for the constants from the ML-Rules model and code to create the initial solution is generated.

```
public class Model implements IModel {

  public static final SolutionFactory solutionFactory$ = new SolutionFactory();
  public static final String aString = "foo";
  public static final double aNumber = ((3 + (4 * 12)) + 2);

  @Override
  public ISolution getInitialSolution() {
    return solutionFactory$.create()
      .add(1, new A(solutionFactory$.create()
        .add((int) 2, new B(aNumber)))
      )
    ;
  }

  // other members omitted
}
```

Each constant is represented as a static final member of the class `Model`. The type of the constant in Java is inferred from its defining expression in ML-Rules. To create the initial solution, an empty solution is created by invoking a `SolutionFactory` (provided by the `mlrules3-common` library). Then, the generated code uses the constructors of the generated species classes to create species instances and add them to the solution. The same approach is taken for subsolutions of the entities to create. The `ISolution` offers a fluent interface, which allows chaining the additions to a solution in one expression.

B.3.2. Generating Java code for an ML-Rules rule

Consider the following ML-Rules model again.

```
A()[];
B(num);

aString: 'foo';
aNumber: 3 + 4 * 12 + 2;

>>INIT[A[2 B(aNumber)]];

A[B(x):b + B(x) + rest]:a -> A[rest] @ #b * (#b - 1);
```

The rule in the model above was used as a high-level example for code-generation in Figure 4. The following listing shows a source code snippet of the code generated for the rule.

```
public class Rule0 implements IRule {

  @Override
  public Iterable<IReaction> match(ISolution solution$) {
    List<IReaction> rs$ = new ArrayList<IReaction>();
```

```
for (A $root$0 : solution$.getEntities(A.class)) {
  final int needed$root$0 = 1;
  final int alreadyTaken$root$0 = 0;
  final int available$root$0 =
    solution$.getCount($root$0) - alreadyTaken$root$0;
  if(available$root$0 < needed$root$0) {
    continue;
  }
  final A a = $root$0;
  final int available$a = available$root$0;
  for (B $root$0$0 : $root$0.subSol.getEntities(B.class)) {
    final int needed$root$0$0 = 1;
    final int alreadyTaken$root$0$0 = 0;
    final int available$root$0$0 =
      $root$0.subSol.getCount($root$0$0) - alreadyTaken$root$0$0;
    if(available$root$0$0 < needed$root$0$0) {
      continue;
    }
    final B b = $root$0$0;
    final int available$b = available$root$0$0;
    final double x = $root$0$0.attribute_0;

    for (B $root$0$1 : $root$0.subSol.getEntities(B.class)) {
      final int needed$root$0$1 = 1;
      final int alreadyTaken$root$0$1 =
        0 + (($root$0$0.equals($root$0$1)) ? needed$root$0$0 : 0);
      final int available$root$0$1 =
        $root$0.subSol.getCount($root$0$1) - alreadyTaken$root$0$1;
      if(available$root$0$1 < needed$root$0$1) {
        continue;
      }
      if(x != $root$0$1.attribute_0) {
        continue;
      }
      ISolution rest = $root$0.subSol.copy();
      rest.remove(needed$root$0$0, $root$0$0);
      rest.remove(needed$root$0$1, $root$0$1);

      rs$.add(new IReaction() {
        @Override
        public double getRate() {
          return (available$b * (available$b - 1));
        }
```

```
            @Override
            public void execute() {
              solution$.remove(needed$root$0, $root$0);
              solution$
                .add(1, new A(solutionFactory$.create()
                  .add(rest))
                )
              ;
            }
          });
        }
      }
    }
    return rs$;
  }
}
```

The code shows the differently indented levels of iteration over the entities in the solution (or the subsolution of previously matched reactants). In each indentation level, that is for each reactant, code for the following steps is generated:

- A `for`-loop iterates all distinct entities of the class of the reactant. The currently considered entity is bound to a variable named `$root$`i_1`$`i_2`$...$`i_n, meaning that the reactant is the i_jth reactant on the jth level. In the following, we refer to this name as `n`.
- Next, some variables that hold information about the number of entities are generated.

  `needed$n` holds the number of entities required by the rule (the reactant count).

  `alreadyTaken$n` holds the number of equal entities that have already been matched by earlier reactants in the same solution. This number can be computed from `n` alone: let `n` be `$root$`i_1`$`i_2`$...$`i_n. Then the value held by the variables `n'` $=$ `$root$`i_1`$`i_2`$...$`i_j with $i_j < i_n$ need to be compared with `n`. For those `n'` where `n'` $=$ `n`, the `needed$n'` add up to `alreadyTaken$n`.

  `available$n` holds the number of entities available in the solution when matching the reactant.
- At this point, an `if`-statement is generated that checks if enough entities are available. If not, the loop continues with the next entity.
- If a variable `v` is assigned to the pattern in the model and this is the first occurrence of the variable `v`, two additional Java variables are generated.
  - `v` is a alias for `n`.
  - `available$v` is an alias for `available$n`.

- If a variable `v` is assigned to the pattern in the model and this is *not* the first occurrence of the variable `v`, an `if`-statement is generated. It compares `n` and `v`. If they are not equal, the loop continues with the next entity.
- Next, each attribute of the reactant pattern is handled.
  - If the pattern is a wildcard, no code is generated.
  - If the pattern is some constant value, an `if`-statement that compares the attribute value of the entity and the constant is generated. If they are not equal, the loop continues with the next entity.
  - If the pattern is a variable and this is the first occurrence of the variable, a Java variable of the same name is generated and initialized with the attribute value of the entity.
  - If the pattern is a variable and this is *not* the first occurrence of the variable, an `if`-statement that compares the attribute value of the entity and the variable is generated. If they are not equal, the loop continues with the next entity.
- If the reactant pattern has a subsolution pattern, the pattern matching continues in the subsolution in the same way.
- Finally, if a list of reactant patterns closes with a rest solution variable, it is handled after all reactant patterns. Similar to other variables, either a variable definition and initialization is generated (if the variable is used for the first time), or an `if`-statement is generated that short-circuits the loop if the rest solution is not equal to the value of the variable.

B.4. Functional simulation experiments

This section contains some technical details regarding the specification of simulation experiments in the functional paradigm in Section 7.3.

B.4.1. Deterministic random number generation

The generation of (pseudo-)random numbers is a frequent textbook example of functional programming, as it nicely illustrates the difference to imperative programming [48, p. 78ff][169, p. 349ff] In imperative programming the state of the random number generator (RNG) is stored in a global variable and implicitly mutated when a random number is drawn. Therefore, the function to draw a random number is not deterministic and, thus, not a pure function:

```
val nextInt: () => Int
nextInt()              // 1449894854
nextInt()              // -1718236964
```

As the RNG state is modified as a side effect of calling the method `nextInt()`, subsequent calls produce different results.

In contrast, drawing a random number in pure functional programming is a deterministic function that explicitly takes the RNG state as an input and outputs the new RNG state together with the drawn value.

```
val rng:     RNG                  // created with some seed
val nextInt: RNG => (RNG, Int)
nextInt(rng)              // (RNG(-7964744663189004623),-2014421327)
nextInt(rng)              // (RNG(-7964744663189004623),-2014421327)
```

To draw several random numbers during a computation, the RNG must be explicitly passed between function calls. This quickly becomes unwieldy.

```
val (r1, rng1) = nextInt(rng)     // (RNG(-7964744663189004623),-2014421327)
val (r2, rng2) = nextInt(rng1)    // (RNG( 4159066171780167020),-968034964)
```

To alleviate that, drawing random numbers can be moved into the `State[S, A]` monad with the RNG as the state `S`. A value of the type `State[S, A]` essentially wraps a computation of the type `S => (S, A)`. Then, threading the state through the computation is handled by the internal wiring of the state monad, and Scala's for-comprehensions can be utilized to succinctly implement functions that rely on a source of randomness.

```
val nextInt:     State[RNG, Int]        = State[RNG, Int](rng => nextInt(rng))
val nextTwoInts: State[RNG, (Int, Int)] = for {
  r1 <- nextInt
  r2 <- nextInt
} yield (r1, r2)
```

Scala implementations of the state monad and related functions are available in libraries for pure functional programming such as cats[1].

B.4.2. Simple experiments

Section 7.3.1 shows some code snippets for expressing a parameter scan with a fixed number of replications per parametrization. These are the functions from these snippets with complete type annotations:

```
def rndSeed: State[RNG, Seed]

def sim(params: Params, seed: Seed): Result

def replications(params: Params, n: Int): State[RNG, List[Result]] =
  rndSeed.map(sim(params, _)).replicateA(n)

def scan(paramss: List[Params], n: Int): State[RNG, Map[Params, IO[List[Result]]]] =
  paramss.traverse(p => replications(p, n).map(p -> _)).map(_.toMap)
```

[1] https://typelevel.org/cats/

B.4.3. Composing the state monad and the `IO` monad

To speed up the execution of experiments, we would like to be able to execute simulation runs in parallel without compromising determinism. One way to parallelize computations in functional programming is wrapping them in a monadic type constructor `F[_]`. Then a value of type `F[A]` describes a computation that, when finished, produces a value of type `A`. Several Scala libraries that provide such an effect monad, for example ZIO[2] or Monix Task[3]. Here, we use the `IO` monad from cats-effect[4].

There are two ways of adding the `IO` monad to the state monad wrapped in our `Experiment` type. The stateful action can return an `IO` value, or the state monad transformer can be used.

```
State [RNG, IO[A]] // wraps RNG => (RNG, IO[A])
StateT[IO, RNG, A] // wraps RNG => IO[(RNG, A)]
```

The decisive difference between both types is that the first one is an applicative functor (`Applicative` in cats), but not a monad, whereas the second one is a monad (the `StateT` monad transformer applied to the `IO` monad).

To illustrate how `State[RNG, IO[_]]` supports parallelization, the following listing shows the definition of the function `ap` in the `Applicative` instance for `State[RNG, IO[_]]`.

```
def ap[A, B](ff: State[S, IO[A => B]])(fa: State[S, IO[A]]): State[S, IO[B]] =
  for {
    f <- ff
    a <- fa
  } yield Parallel.parAp(f)(a)
```

Given two `State[RNG, IO[_]]` values, we first *sequentially* produce the `IO` values `f` and `a`. The `IO` values are evaluated *in parallel* and then combined by the library function `parAp`. In our experiment setting, we can use this to sequentially generate seeds (which typically is fast) for the simulation runs and then execute the runs in parallel (which typically dominates the overall run time).

The functions `replications` and `scan` from above can be embedded into the applicative functor as follows:

```
type App[T] = State[RNG, IO[T]]
val app: Applicative[App]

def replications(params: Params, n: Int): State[RNG, IO[List[Result]]] =
  app.replicateA(n, rndSeed.map(s => IO.delay(sim(params, s))))

def scan(paramss: List[Params], n: Int): State[RNG, IO[Map[Params, List[Result]]]] =
  app.map(paramss.traverse[App, (Params, List[Result])](p => app.map(replications(p, n))
                                                                  (p -> _)))
```

[2] https://zio.dev/
[3] https://monix.io/api/3.0/monix/eval/Task.html
[4] https://github.com/typelevel/cats-effect

```
        (_.toMap)
```

Note that the same combinators (`replicateA`, `traverse`) as above are used to express the structure of the computation. However, we need to explicitly summon the `Applicative` instance `app` and add some type annotations. Otherwise, Scala's type inference prefers the `Applicative` instance for `State` over our custom `Applicative` for `State[RNG, IO[_]]`.

To chain computations, the monadic type `StateT[IO, RNG, _]` can be used. Here, the wrapped function is of type `RNG => IO[(RNG, _)]` and produces an `IO` value that takes ownership of the RNG until it is finished. Thus, it is not possible to execute multiple `StateT[IO, RNG, _]` in parallel. To combine parallelization and chaining, a parallel batch of runs can be defined as a `State[RNG, IO[_]]` and then used as subcomputation in a a `StateT[IO, RNG, _]`. To do that, it must be converted from the applicative functorial type to the monadic type, which can be done as follows:

```
def toStateT[S, A](sfa: State[S, IO[A]]): StateT[IO, S, A] =
  StateT[IO, S, A] { s =>
    val (s1, fa) = sfa.run(s).value
    fa.map((s1, _))
  }
```

Essentially, the random number generator `s` is used to start the inner computation and then lifted into it after it finishes.

Using this conversion, dynamic replication conditions can be implemented for the monadic type `StateT[IO, RNG, _]` by using the library function `iterateUntilM`. Starting with an empty list, batches are executed and concatenated until the replication condition is true when evaluated on the list of results so far:

```
def replicationsUntil[A](batch: StateT[IO, RNG, List[A]],
                      replCond: List[A] => Boolean)
                      : StateT[IO, RNG, List[A]] =
  List.empty[A].iterateUntilM(results => batch.map(_ ++ results))(replCond)
```

Composing these functions yields the implementation of the statistical model-checking experiment shown in Section 7.3.5. In the listing below we omitted some of the hypothesis testing code, which evaluates some mathematical expressions based on the statistical parameters `p`, `alpha`, `beta`, and `delta`.

```
def sequentialProbabilityRatioTest(prop: Result => Boolean,
                                   batch: State[RNG, IO[List[Result]]],
                                   p: Double,
                                   alpha: Double,
                                   beta: Double,
                                   delta: Double
                                  ): StateT[RNG, IO, Boolean] = {

  def result(completed: List[Boolean]): Option[Boolean] = /* omitted */
  def enoughRuns(completed: List[Boolean]): Boolean =
    result(completed).isDefined

  val checkedBatch: StateT[IO, RNG, List[Boolean]] = toStateT(batch).map(_.map(prop))
  replicationsUntil(checkedBatch, enoughRuns).map(result(_).get)
}
```

The function `result` returns the result of the hypothesis test on the simulation runs completed so far. Three results are possible: `None`, when more runs are needed, or `Some(true)` or `Some(false)` when the property is satisfied or not satisfied, respectively. Based on `result`, `enoughRuns` determines if more runs are needed and is used as replication condition in the last line. Before that, `checkedBatch` represents a batch of runs on which the property `prop` has been evaluated to **`true`** or **`false`**. Such batches are executed until `enoughRuns` returns true, and the overall result is returned.

Bibliography

[1] Paul W. Abrahams. “A final solution to the Dangling else of ALGOL 60 and related languages”. In: *Communications of the ACM* 9.9 (Sept. 1966), pp. 679–682. DOI: 10.1145/365813.365821.

[2] Gul Agha and Karl Palmskog. “A Survey of Statistical Model Checking”. In: *ACM Transactions on Modeling and Computer Simulation* 28.1 (2018), 6:1–6:39. ISSN: 1049-3301. DOI: 10.1145/3158668.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986. ISBN: 0-201-10088-6.

[4] Linda J. S. Allen and Glenn E. Lahodny. “Extinction Thresholds in Deterministic and Stochastic Epidemic Models”. In: *Journal of Biological Dynamics* 6 (2012), pp. 590–611. DOI: 10.1080/17513758.2012.665502.

[5] Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. “On the Power of Attribute-Based Communication”. In: *Formal Techniques for Distributed Objects, Components, and Systems.* Ed. by Elvira Albert and Ivan Lanese. Springer International Publishing, 2016, pp. 1–18. DOI: 10.1007/978-3-319-39570-8_1.

[6] William J. Anderson. *Continuous-Time Markov Chains: An Applications-Oriented Approach (Springer Series in Statistics).* Springer, 1991. ISBN: 0-387-97369-9.

[7] Cyrille Artho et al. “Domain-Specific Languages with Scala”. In: *Formal Methods and Software Engineering.* Ed. by Michael Butler, Sylvain Conchon, and Fatiha Zaïdi. Springer International Publishing, 2015, pp. 1–16. DOI: 10.1007/978-3-319-25423-4_1.

[8] *AssertJ / Fluent assertions for java.* 2018. URL: https://joel-costigliola.github.io/assertj/ (visited on 11/29/2018).

[9] Douglas M. Auclair. “MonadPlus: What a Super Monad!” In: *The Monad Reader* 11 (2008). URL: https://wiki.haskell.org/wikiupload/6/6a/TMR-Issue11.pdf.

[10] Franz Baader, Tobias Nipkow, and Baader Franz. *Term Rewriting and All That.* Cambridge University Press, 2006. 316 pp. ISBN: 0521779200.

[11] Leo Bachmair, Ta Chen, and I. V. Ramakrishnan. “Associative-commutative discrimination nets”. In: *Lecture Notes in Computer Science.* Ed. by M. C. Gaudel and J. P. Jouannaud. Springer Berlin Heidelberg, 1993, pp. 61–74. DOI: 10.1007/3-540-56610-4_56.

[12] Monya Baker. “1,500 scientists lift the lid on reproducibility”. In: *Nature* 533.7604 (May 2016), pp. 452–454. DOI: 10.1038/533452a.

[13] Osman Balci. “A life cycle for modeling and simulation”. In: *SIMULATION* 88.7 (Feb. 2012), pp. 870–883. DOI: 10.1177/0037549712438469.

[14] Jerry Banks et al. *Discrete-Event System Simulation*. 5th ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2010. ISBN: 0138150370.

[15] Howard Barringer and Klaus Havelund. “Internal versus External DSLs for Trace Analysis”. In: *Runtime Verification*. Ed. by Sarfraz Khurshid and Koushik Sen. Springer Berlin Heidelberg, 2012, pp. 1–3. DOI: 10.1007/978-3-642-29860-8_1.

[16] Howard Barringer and Klaus Havelund. “TraceContract: A Scala DSL for Trace Analysis”. In: *Lecture Notes in Computer Science*. Ed. by Michael Butler and Wolfram Schulte. Springer Berlin Heidelberg, 2011, pp. 57–72. DOI: 10.1007/978-3-642-21437-0_7.

[17] Andreea Beica, Calin C. Guet, and Tatjana Petrov. “Efficient Reduction of Kappa Models by Static Inspection of the Rule-Set”. In: *Hybrid Systems Biology*. Ed. by Alessandro Abate and David Šafránek. Springer International Publishing, 2015, pp. 173–191. DOI: 10.1007/978-3-319-26916-0_10.

[18] Dan Benanav, Deepak Kapur, and Paliath Narendran. “Complexity of matching problems”. In: *Journal of Symbolic Computation* 3.1-2 (Feb. 1987), pp. 203–216. DOI: 10.1016/s0747-7171(87)80027-5.

[19] Jon Bentley. “Programming pearls”. In: *Communications of the ACM* 29.8 (Aug. 1986), pp. 711–721. DOI: 10.1145/6424.315691.

[20] Anja Bethge, Udo Schumacher, and Gero Wedemann. “Simulation of metastatic progression using a computer model including chemotherapy and radiation therapy”. In: *Journal of Biomedical Informatics* 57 (2015), pp. 74–87. ISSN: 1532-0464. DOI: 10.1016/j.jbi.2015.07.011.

[21] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend*. 2nd ed. Packt Publishing, 2016. ISBN: 1786464969.

[22] Lachlan Birdsey, Claudia Szabo, and Katrina Falkner. “CASL: A declarative domain specific language for modeling Complex Adaptive Systems”. In: *Proceedings of the 2016 Winter Simulation Conference*. IEEE, 2016. DOI: 10.1109/wsc.2016.7822180.

[23] Rúnar Bjarnason. *Constraints Liberate, Liberties Constrain*. [Video]. 2016. URL: https://www.youtube.com/watch?v=GqmsQeSzMdw.

[24] Bruno Blanchet. “Escape analysis for JavaTM”. In: *ACM Transactions on Programming Languages and Systems* 25.6 (Nov. 2003), pp. 713–775. DOI: 10.1145/945885.945886.

[25] Michael L. Blinov et al. “BioNetGen: Software for Rule-based Modeling of Signal Transduction Based on the Interactions of Molecular Domains”. In: *Bioinformatics* 20.17 (2004), pp. 3289–3291. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bth378.

[26] Michael L. Blinov et al. “Graph Theory for Rule-Based Modeling of Biochemical Networks”. In: *Lecture Notes in Computer Science*. Ed. by Corrado Priami et al. Springer Berlin Heidelberg, 2006, pp. 89–106. DOI: 10.1007/11905455_5.

[27] Joshua Bloch. *Effective Java*. Addison-Wesley Professional, 2017.

[28] Denis Bogdanas and Grigore Roşu. “K-Java: A Complete Semantics of Java”. In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’15. Mumbai, India: ACM, 2015, pp. 445–456. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676982.

[29] Ronald F. Boisvert. “Incentivizing reproducibility”. In: *Communications of the ACM* 59.10 (Sept. 2016), pp. 5–5. DOI: 10.1145/2994031.

[30] Luca Bortolussi, Guido Sanguinetti, and Simone Silvetti. “Bayesian statistical parametric verification and synthesis by machine learning”. In: *Proceedings of the 2018 Winter Simulation Conference*. IEEE, 2018. DOI: 10.1109/wsc.2018.8632443.

[31] Luca Bortolussi et al. “CARMA: Collective Adaptive Resource-sharing Markovian Agents”. In: *Quantitative Aspects of Programming Languages and Systems (QAPL)*. Vol. 194. 2015, pp. 16–31. DOI: 10.4204/EPTCS.194.2.

[32] Pierre Boutillier, Ioana Cristescu, and Jérôme Feret. “Counters in Kappa: Semantics, Simulation, and Static Analysis”. In: *Programming Languages and Systems*. Ed. by Luís Caires. Cham: Springer International Publishing, 2019, pp. 176–204. ISBN: 978-3-030-17184-1. DOI: 10.1007/978-3-030-17184-1_7.

[33] Pierre Boutillier et al. “KaSa: A Static Analyzer for Kappa”. In: *Computational Methods in Systems Biology*. Springer International Publishing, 2018, pp. 285–291. DOI: 10.1007/978-3-319-99429-1_17.

[34] Pierre Boutillier et al. *The Kappa Language and Kappa Tools*. Mar. 23, 2020. URL: http://www.kappalanguage.org.

[35] Pierre Boutillier et al. “The Kappa platform for rule-based modeling”. In: *Bioinformatics* 34.13 (June 2018), pp. i583–i592. DOI: 10.1093/bioinformatics/bty272.

[36] Edwin C. Brady and Kevin Hammond. “Scrapping your inefficient engine”. In: *ACM SIGPLAN Notices* 45.9 (Sept. 2010), pp. 297–308. DOI: 10.1145/1932681.1863587.

[37] Frederick P. Brooks. “No Silver Bullet Essence and Accidents of Software Engineering”. In: *Computer* 20.4 (1987), pp. 10–19. DOI: 10.1109/MC.1987.1663532.

[38] Randy Brown. “Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem”. In: *Communications of the ACM* 31.10 (1988), pp. 1220–1227. DOI: 10.1145/63039.63045.

[39] Kai Budde et al. “Exploiting equation-free analysis for multi-level, agent-based models in cell biology”. In: *Proceedings of the 2017 Winter Simulation Conference*. IEEE, 2017. DOI: 10.1109/wsc.2017.8248206.

[40] Hendrik Bünder. "Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages". In: *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*. SCITEPRESS - Science and Technology Publications, 2019. DOI: 10.5220/0007556301290140.

[41] Donald T. Campbell. "'Downward Causation' in Hierarchically Organised Biological Systems". In: *Studies in the Philosophy of Biology*. Ed. by Francisco Jose Ayala and Theodosius Dobzhansky. Macmillan Education UK, 1974, pp. 179–186. DOI: 10.1007/978-1-349-01892-5_11.

[42] Yang Cao, Hong Li, and Linda Petzold. "Efficient formulation of the stochastic simulation algorithm for chemically reacting systems". In: *The Journal of Chemical Physics* 121.9 (Sept. 2004), pp. 4059–4067. DOI: 10.1063/1.1778376.

[43] Luca Cardelli. "From processes to ODEs by chemistry". In: *Proceedings of the Fifth IFIP International Conference On Theoretical Computer Science*. Springer. 2008, pp. 261–281. DOI: 10.1007/978-0-387-09680-3_18.

[44] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages". In: *Journal of Functional Programming* 19.5 (Sept. 2009), pp. 509–543. ISSN: 0956-7968. DOI: 10.1017/S0956796809007205.

[45] Francois E. Cellier. *Continuous System Modeling*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1991.

[46] Stefano Ceri and Georg Gottlob. "Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries". In: *IEEE Transactions on Software Engineering* SE-11.4 (Apr. 1985), pp. 324–345. ISSN: 0098-5589. DOI: 10.1109/TSE.1985.232223.

[47] Souti Chattopadhyay et al. "What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities". In: *Proceedings of the CHI'20*. 2020. DOI: 10.1145/3313831.3376729.

[48] Paul Chiusano and Runar Bjarnason. *Functional Programming in Scala*. Manning Publications, Sept. 25, 2014. 320 pp. ISBN: 1617290653.

[49] Noam Chomsky. "On certain formal properties of grammars". In: *Information and Control* 2.2 (June 1959), pp. 137–167. DOI: 10.1016/s0019-9958(59)90362-6.

[50] Leonardo Chwif, Marcos Ribeiro Pereira Barretto, and Ray J. Paul. "On simulation model complexity". In: *Proceedings of the 2000 Winter Simulation Conference*. IEEE, 2000. DOI: 10.1109/wsc.2000.899751.

[51] Claudio Cioffi-Revilla and Mark Rouleau. "MASON RebeLand: An Agent-Based Model of Politics, Environment, and Insurgency". In: *International Studies Review* 12.1 (2010), pp. 31–52. DOI: 10.1111/j.1468-2486.2009.00911.x.

[52] *Clang - Features and Goals*. 2018. URL: https://clang.llvm.org/features.html (visited on 11/01/2018).

[53] John R. Clymer. *Simulation-based engineering of complex systems*. 2nd ed. USA: Wiley-Interscience, 2009. ISBN: 0-470-40129-X.

[54] Association for Computing Machinery. *Artifact Review and Badging*. Apr. 1, 2018. URL: https://www.acm.org/publications/policies/artifact-review-badging.

[55] H. Conrad Cunningham. "A little language for surveys". In: *Proceedings of the 46th Annual Southeast Regional Conference on XX - ACM-SE 46*. ACM Press, 2008. DOI: 10.1145/1593105.1593181.

[56] Vincent Danos and Cosimo Laneve. "Formal molecular biology". In: *Theoretical Computer Science*. Computational Systems Biology 325.1 (2004), pp. 69–110. DOI: 10.1016/j.tcs.2004.03.065.

[57] Tuğrul Dayar et al. "Bounding the equilibrium distribution of Markov population models". In: *Numerical Linear Algebra with Applications* 18.6 (Oct. 2011), pp. 931–946. DOI: 10.1002/nla.795.

[58] Rocco De Nicola et al. "A uniform definition of stochastic process calculi". In: *ACM Computing Surveys* 46.1 (Oct. 2013), pp. 1–35. DOI: 10.1145/2522968.2522973.

[59] Rocco De Nicola et al. "AErlang: Empowering Erlang with Attribute-Based Communication". In: *Lecture Notes in Computer Science*. Ed. by Jean-Marie Jacquet and Mieke Massink. Springer International Publishing, 2017, pp. 21–39. DOI: 10.1007/978-3-319-59746-1_2.

[60] Rocco De Nicola et al. "Rate-Based Transition Systems for Stochastic Process Calculi". In: *Automata, Languages and Programming*. Ed. by Susanne Albers et al. Vol. 5556. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 435–446. ISBN: 978-3-642-02929-5. DOI: 10.1007/978-3-642-02930-1_36. (Visited on 06/16/2020).

[61] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-specific Languages: An Annotated Bibliography". In: *SIGPLAN Not.* 35.6 (June 2000), pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035.

[62] Joseph L. Doob. "Markoff Chains–Denumerable Case". In: *Transactions of the American Mathematical Society* 58.3 (1945), p. 455. DOI: 10.1090/s0002-9947-1945-0013857-4.

[63] Sven Efftinge and Markus Voelter. "oAW xText: A framework for textual DSLs". In: *Eclipse Modeling Symposium at Eclipse Summit Europe 2006*. 2006. URL: http://voelter.de/data/workshops/EfftingeVoelterEclipseSummit.pdf.

[64] Sven Efftinge et al. "Xbase: implementing domain-specific languages for Java". In: *ACM SIGPLAN Notices* 48.3 (Apr. 2013), pp. 112–121. DOI: 10.1145/2480361.2371419.

[65] J. Elf and M. Ehrenberg. "Spontaneous Separation of Bi-Stable Biochemical Systems Into Spatial Domains of Opposite Phases". In: *Systems Biology* 1 (2 2004), pp. 230–236. DOI: 10.1049/sb:20045021.

[66] Conal Elliott. "Compiling to categories". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (Aug. 2017), pp. 1–27. DOI: 10.1145/3110271.

[67] Chucky Ellison and Grigore Rosu. “An Executable Formal Semantics of C with Applications”. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’12. Philadelphia, PA, USA: ACM, 2012, pp. 533–544. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103719.

[68] Radek Erban and S. Jonathan Chapman. “Stochastic modelling of reaction–diffusion processes: algorithms for bimolecular reactions”. In: *Physical Biology* 6.4 (Aug. 2009), p. 046001. DOI: 10.1088/1478-3975/6/4/046001.

[69] Sebastian Erdweg et al. “The State of the Art in Language Workbenches”. In: *Software Language Engineering*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Cham: Springer International Publishing, 2013, pp. 197–217. ISBN: 978-3-319-02654-1. DOI: 10.1007/978-3-319-02654-1_11.

[70] Roland Ewald and Adelinde M. Uhrmacher. “SESSL: A Domain-specific Language for Simulation Experiments”. In: *ACM Transactions on Modeling and Computer Simulation* 24.2 (2014), 11:1–11:25. ISSN: 1049-3301. DOI: 10.1145/2567895.

[71] James R. Faeder, Michael L. Blinov, and William S. Hlavacek. “Rule-Based Modeling of Biochemical Systems with BioNetGen”. In: *Methods in Molecular Biology*. Ed. by Ivan V. Maly. Humana Press, 2009, pp. 113–167. DOI: 10.1007/978-1-59745-525-1_5.

[72] James R. Faeder et al. “Investigation of early events in FcεRI-Mediated signaling using a detailed mathematical model”. In: *The Journal of Immunology* 170.7 (2003), pp. 3769–3781. ISSN: 0022-1767. DOI: 10.4049/jimmunol.170.7.3769.

[73] James R. Faeder et al. “Rule-based modeling of biochemical networks”. In: *Complexity* 10.4 (2005), pp. 22–41. DOI: 10.1002/cplx.20074.

[74] William M. Farmer. “Chiron: A Multi-Paradigm Logic”. In: *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec, Studies in Logic, Grammar and Rhetoric* 10.23 (2007), pp. 1–19. ISSN: 0860-150X. URL: http://logika.uwb.edu.pl/studies/download.php?volid=23&artid=wf&format=PDF.

[75] Matthias Felleisen. “On the expressive power of programming languages”. In: *Science of Computer Programming* 17.1 (1991), pp. 35–75. ISSN: 0167-6423. DOI: 10.1016/0167-6423(91)90036-W.

[76] Cheng Feng and Jane Hillston. “PALOMA: A Process Algebra for Located Markovian Agents”. In: *Quantitative Evaluation of Systems*. Ed. by Gethin Norman and William Sanders. Springer International Publishing, 2014, pp. 265–280. DOI: 10.1007/978-3-319-10696-0_22.

[77] Jasmin Fisher, David Harel, and Thomas A. Henzinger. “Biology as Reactivity”. In: *Communications of the ACM* 54.10 (Oct. 2011), pp. 72–82. ISSN: 0001-0782. DOI: 10.1145/2001269.2001289.

[78] Jasmin Fisher and Thomas A. Henzinger. “Executable cell biology”. en. In: *Nature Biotechnology* 25.11 (2007), pp. 1239–1249. DOI: 10.1038/nbt1356.

[79] Paul A. Fishwick. “A taxonomy for simulation modeling based on programming language principles”. In: *IIE Transactions* 30.9 (Sept. 1, 1998), pp. 811–820. ISSN: 1573-9724. DOI: 10.1023/A:1007548116679.

[80] Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

[81] Robert France and Bernhard Rumpe. “Domain specific modeling”. In: *Software & Systems Modeling* 4.1 (Feb. 1, 2005), pp. 1–3. ISSN: 1619-1374. DOI: 10.1007/s10270-005-0078-1.

[82] Robert France and Bernhard Rumpe. “Model-driven Development of Complex Software: A Research Roadmap”. In: *2007 Future of Software Engineering.* FOSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–54. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.14.

[83] Peter Fritzson and Peter Bunus. “Modelica - a general object-oriented language for continuous and discrete-event system modeling and simulation”. In: *Proceedings of the 35th Annual Simulation Symposium. SS 2002.* Apr. 2002, pp. 365–380. DOI: 10.1109/SIMSYM.2002.1000174.

[84] Yoshihiko Futamura. “Partial Evaluation of Computation Process—AnApproach to a Compiler-Compiler”. In: *Higher Order Symbol. Comput.* 12.4 (Dec. 1999), pp. 381–391. ISSN: 1388-3690. DOI: 10.1023/A:1010095604496.

[85] Paolo G. Giarrusso and Jonathan Immanuel Brachthäuser. “Revisiting the Cake Pattern: Scaling “Scalable Component Abstractions””. In: *Unpublished draft accompanying a talk at Scala Symposium.* Vol. 1. 2016/10. 2016, p. 24. URL: https://files.b-studios.de/revisiting-cake-pattern.pdf.

[86] Jeremy Gibbons and Nicolas Wu. “Folding domain-specific languages”. In: *ACM SIGPLAN Notices* 49.9 (Aug. 2014), pp. 339–347. DOI: 10.1145/2692915.2628138.

[87] Michael A. Gibson and Jehoshua Bruck. “Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels”. In: *The Journal of Physical Chemistry A* 104.9 (2000), pp. 1876–1889. DOI: 10.1021/jp993732q.

[88] Daniel T. Gillespie. “A general method for numerically simulating the stochastic time evolution of coupled chemical reactions”. In: *Journal of Computational Physics* 22.4 (1976), pp. 403–434. ISSN: 0021-9991. DOI: 10.1016/0021-9991(76)90041-3.

[89] Daniel T. Gillespie. “Exact stochastic simulation of coupled chemical reactions”. In: *The Journal of Physical Chemistry* 81.25 (1977), pp. 2340–2361. DOI: 10.1021/j100540a008.

[90] Daniel T. Gillespie. *Markov Processes: An Introduction for Physical Scientists.* ACADEMIC PR INC, Oct. 1, 1991. 592 pp. ISBN: 0122839552.

[91] Daniel T. Gillespie. “Stochastic Simulation of Chemical Kinetics”. In: *Annual Review of Physical Chemistry* 58.1 (May 2007), pp. 35–55. DOI: 10.1146/annurev.physchem.58.032806.104637.

[92] J. A. Goguen et al. “Initial Algebra Semantics and Continuous Algebras”. In: *Journal of the ACM (JACM)* 24.1 (Jan. 1977), pp. 68–95. DOI: 10.1145/321992.321997.

[93] Volker Grimm and Steven F. Railsback. *Individual-based modeling and ecology*. Princeton university press, 2013. DOI: 10.1515/9781400850624.

[94] Abhishekh Gupta and Pedro Mendes. “An Overview of Network-Based and -Free Approaches for Stochastic Simulation of Biochemical Systems”. In: *Computation* 6.1 (2018). ISSN: 2079-3197. DOI: 10.3390/computation6010009.

[95] Fiete Haack et al. “Spatio-temporal Model of Endogenous ROS and Raft-Dependent WNT/Beta-Catenin Signaling Driving Cell Fate Commitment in Human Neural Progenitor Cells”. In: *PLOS Computational Biology* 11.3 (Mar. 2015), pp. 1–28. DOI: 10.1371/journal.pcbi.1004106.

[96] Kevin Hammond. “Why Parallel Functional Programming Matters: Panel Statement”. In: *Reliable Software Technologies - Ada-Europe 2011*. Springer Berlin Heidelberg, 2011, pp. 201–205. DOI: 10.1007/978-3-642-21338-0_17.

[97] Leonard Harris, Justin S. Hogg, and James R. Faeder. “Compartmental rule-based modeling of biochemical systems”. In: *Proceedings of the 2009 Winter Simulation Conference*. WSC ’09. Austin, Texas: IEEE, 2009. DOI: 10.1109/WSC.2009.5429719.

[98] Tobias Helms, Tom Warnke, and Adelinde M. Uhrmacher. “Multi-Level Modeling and Simulation of Cellular Systems: An Introduction to ML-Rules”. In: *Modeling Biomolecular Site Dynamics*. Ed. by William S. Hlavacek. Springer New York, 2019, pp. 141–160. DOI: 10.1007/978-1-4939-9102-0_6.

[99] Tobias Helms et al. “Automatic Runtime Adaptation for Component-Based Simulation Algorithms”. In: *ACM Transactions on Modeling and Computer Simulation* 26.1 (2015), 7:1–7:24. DOI: 10.1145/2821509.

[100] Tobias Helms et al. “Multi-level modeling and simulation of cell biological systems with ML-Rules - A tutorial”. In: *Proceedings of the 2014 Winter Simulation Conference*. IEEE, 2014. DOI: 10.1109/wsc.2014.7019887.

[101] Tobias Helms et al. “Semantics and Efficient Simulation Algorithms of an Expressive Multi-Level Modeling Language”. In: *ACM Transactions on Modeling and Computer Simulation* 27.2 (May 2017), 8:1–8:25. DOI: 10.1145/2998499.

[102] Tobias Helms et al. “Toward a Language for the Flexible Observation of Simulations”. In: *Proceedings of the 2012 Winter Simulation Conference*. WSC ’12. Berlin, Germany: IEEE, 2012. DOI: 10.1109/wsc.2012.6465073.

[103] Thomas A. Henzinger, Barbara Jobstmann, and Verena Wolf. “Formalisms for Specifying Markovian Population Models”. In: *International Journal of Foundations of Computer Science* 22.4 (2011), pp. 823–841. ISSN: 0129-0541. DOI: 10.1007/978-3-642-04420-5_2.

[104] Christoph Herrmann et al. “An Algebraic View on the Semantics of Model Composition”. In: *Model Driven Architecture- Foundations and Applications*. Ed. by David H. Akehurst, Régis Vogel, and Richard F. Paige. Springer Berlin Heidelberg, 2007, pp. 99–113. DOI: 10.1007/978-3-540-72901-3_8.

[105] Justin S. Hogg et al. “Exact Hybrid Particle/Population Simulation of Rule-Based Models of Biochemical Systems”. In: *PLOS Computational Biology* 10.4 (Apr. 2014), pp. 1–16. DOI: 10.1371/journal.pcbi.1003544.

[106] Ricardo Honorato-Zimmer et al. “Chromar, a language of parameterised agents”. In: *Theoretical Computer Science* 765 (Apr. 2019), pp. 97–119. DOI: 10.1016/j.tcs.2017.07.034.

[107] Michael Hucka et al. “The systems biology markup language (SBML): A medium for representation and exchange of biochemical network models”. In: *Bioinformatics* 19.4 (Mar. 2003), pp. 524–531. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btg015.

[108] Graham Hutton and Erik Meijer. *Monadic parser combinators*. Technical Report NOTTCS-TR-96-4. University of Nottingham, 1996. URL: http://eprints.nottingham.ac.uk/237/.

[109] Graham Hutton and Erik Meijer. “Monadic Parsing in Haskell”. In: *Journal of Functional Programming* 8.4 (July 1998), pp. 437–444. ISSN: 0956-7968. DOI: 10.1017/S0956796898003050.

[110] Peter Ivie and Douglas Thain. “Reproducibility in Scientific Computing”. In: *ACM Computing Surveys* 51.3 (July 2018), pp. 1–36. DOI: 10.1145/3186266.

[111] Peter Ivie et al. “An analysis of reproducibility and non-determinism in HEP software and ROOT data”. In: *Journal of Physics: Conference Series* 898.10 (Oct. 2017), p. 102007. DOI: 10.1088/1742-6596/898/10/102007.

[112] *java.util.stream (Java Platform SE 8)*. 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html (visited on 11/29/2018).

[113] Mathias John et al. “Biochemical Reaction Rules with Constraints”. In: *Proceedings of the 20th European Symposium on Programming, ESOP 2011*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 338–357. DOI: 10.1007/978-3-642-19718-5_18.

[114] Mathias John et al. “The Attributed Pi Calculus”. In: *Proceedings of the CMSB 2008*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 83–102. DOI: 10.1007/978-3-540-88562-7_10.

[115] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. “Combinators for impure yet hygienic code generation”. In: *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation - PEPM ’14*. ACM Press, 2014. DOI: 10.1145/2543728.2543740.

[116] Jonathan R. Karr et al. “A Whole-Cell Computational Model Predicts Phenotype from Genotype”. In: *Cell* 150.2 (July 2012), pp. 389–401. DOI: 10.1016/j.cell.2012.05.044.

[117] Gabriele Keller et al. “Specialising Simulator Generators for High-performance Monte-Carlo Methods”. In: *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages.* PADL’08. San Francisco, CA, USA: Springer-Verlag, 2008, pp. 116–132. ISBN: 3-540-77441-6. DOI: 10.1007/978-3-540-77442-6_9.

[118] Anna Klabunde et al. *An Agent-Based Decision Model of Migration, Embedded in the Life Course - Model Description in ODD+D Format.* MPIDR Working Paper WP-2015-002. Max Planck Institute for Demographic Research, 2015. URL: http://www.demogr.mpg.de/papers/working/wp-2015-002.pdf.

[119] Jack P. C. Kleijnen et al. “State-of-the-Art Review: A User’s Guide to the Brave New World of Designing Simulation Experiments”. In: *INFORMS Journal on Computing* 17.3 (Aug. 2005), pp. 263–289. DOI: 10.1287/ijoc.1050.0136.

[120] Anneke Kleppe. “A Language Description is More than a Metamodel”. In: *Fourth International Workshop on Software Language Engineering.* 2007. URL: https://research.utwente.nl/en/publications/a-language-description-is-more-than-a-metamodel(21b95490-024b-4713-bb4b-5f7bfbb6022c).html.

[121] Anders Klevmarken. “Dynamic Microsimulation for Policy Analysis: Problems and Solutions”. In: *Simulating an Ageing Population: A Microsimulation Approach Applied to Sweden.* Ed. by Anders Klevmarken and Björn Lindgren. Vol. 285. Contributions to Economic Analysis. Elsevier, 2008, pp. 31–53.

[122] Eugene Kohlbecker et al. “Hygienic macro expansion”. In: *Proceedings of the 1986 ACM conference on LISP and functional programming - LFP ’86.* ACM Press, 1986. DOI: 10.1145/319838.319859.

[123] Christina Kossow et al. “Evaluating Different Modeling Languages Based on a User Study”. In: *Proceedings of the 49th Annual Simulation Symposium.* ANSS ’16. Pasadena, California: Society for Computer Simulation International, 2016. ISBN: 9781510823167. DOI: 10.5555/2962374.2962392.

[124] Till Köster, Tom Warnke, and Adelinde M. Uhrmacher. “Partial Evaluation via Code Generation for Static Stochastic Reaction Network Models”. In: *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.* Miami FL Spain: ACM, June 15, 2020, pp. 159–170. ISBN: 978-1-4503-7592-4. DOI: 10.1145/3384441.3395983. (Visited on 07/02/2020).

[125] Holger Krahn, Bernhard Rumpe, and Steven Völkel. “MontiCore: a framework for compositional development of domain specific languages”. In: *International Journal on Software Tools for Technology Transfer* 12.5 (Mar. 2010), pp. 353–372. DOI: 10.1007/s10009-010-0142-1.

[126] Jeff Kramer. “Is abstraction the key to computing?” In: *Communications of the ACM* 50.4 (Apr. 2007), pp. 36–42. DOI: 10.1145/1232743.1232745.

[127] Manuel Krebber. “Non-linear Associative-Commutative Many-to-One Pattern Matching with Sequence Variables”. In: *CoRR* abs/1705.00907 (2017). arXiv: 1705.00907.

[128] Manuel Krebber, Henrik Barthels, and Paolo Bientinesi. “Efficient Pattern Matching in Python”. In: *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing - PyHPC’17*. ACM Press, 2017. DOI: 10.1145/3149869.3149871.

[129] Thomas G. Kurtz. “The Relationship between Stochastic and Deterministic Models for Chemical Reactions”. In: *The Journal of Chemical Physics* 57.7 (Oct. 1972), pp. 2976–2978. DOI: 10.1063/1.1678692.

[130] Peter J. Landin. “The next 700 programming languages”. In: *Communications of the ACM* 9.3 (Mar. 1966), pp. 157–166. DOI: 10.1145/365230.365257.

[131] Juan de Lara and Hans Vangheluwe. “AToM3: A Tool for Multi-formalism and Meta-modelling”. In: *Fundamental Approaches to Software Engineering*. Ed. by Ralf-Detlef Kutsche and Herbert Weber. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 174–188. ISBN: 978-3-540-45923-1. DOI: 10.1007/3-540-45923-5_12.

[132] Nicolas Laurent and Kim Mens. “Taming context-sensitive languages with principled stateful parsing”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering - SLE 2016*. ACM Press, 2016. DOI: 10.1145/2997364.2997370.

[133] Averill Law. *Simulation Modeling and Analysis*. 4th ed. McGraw-Hill, 2006.

[134] Justin Le. *Unique sample drawing & searches with List and StateT — "Send more money"*. 2015. URL: https://blog.jle.im/entry/unique-sample-drawing-searches-with-list-and-statet.html (visited on 06/25/2019).

[135] Nicolas Le Novere et al. “The systems biology graphical notation”. In: *Nature biotechnology* 27.8 (2009), p. 735. DOI: 10.1038/nbt.1558.

[136] Ethan Lee et al. “The roles of APC and Axin derived from experimental and theoretical analysis of the Wnt pathway”. In: *PLoS biology* 1.1 (2003). DOI: 10.1371/journal.pbio.0000010.

[137] Axel Legay, Benoît Delahaye, and Saddek Bensalem. “Statistical Model Checking: An Overview”. In: *Runtime Verification*. Ed. by Howard Barringer et al. Berlin, Heidelberg: Springer, 2010, pp. 122–135. ISBN: 978-3-642-16612-9. DOI: 10.1007/978-3-642-16612-9_11.

[138] John Levine. *flex & bison*. O’Reilly Media, Inc, USA, Aug. 28, 2009. 271 pp. ISBN: 0596155972.

[139] Stefan Leye, Jan Himmelspach, and Adelinde M. Uhrmacher. “A Discussion on Experimental Model Validation”. In: *Proceedings of the UKSim 2009: 11th International Conference on Computer Modelling and Simulation*. UKSIM ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 161–167. ISBN: 978-0-7695-3593-7. DOI: 10.1109/UKSIM.2009.20.

[140] Mieszko Lis et al. "Efficient stochastic simulation of reaction–diffusion processes via direct compilation". In: *Bioinformatics* 25.17 (July 2009), pp. 2289–2291. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btp387.

[141] Fei Liu and Monika Heiner. "Petri Nets for Modeling and Analyzing Biochemical Reaction Networks". In: *Approaches in Integrative Bioinformatics*. Ed. by Ming Chen and Ralf Hofestädt. Springer Berlin Heidelberg, Oct. 2013, pp. 245–272. DOI: 10.1007/978-3-642-41281-3_9.

[142] Carlos F. Lopez et al. "Programming biological models in Python using PySB". In: *Molecular Systems Biology* 9.1 (2013). ISSN: 1744-4292. DOI: 10.1038/msb.2013.1.

[143] Thomas W. Lucas et al. "Changing the paradigm: Simulation, now a method of first resort". In: *Naval Research Logistics (NRL)* 62.4 (2015), pp. 293–303. DOI: 10.1002/nav.21628.

[144] Martin Lukasiewycz et al. "Opt4J - A Modular Framework for Meta-heuristic Optimization". In: *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*. Dublin, Ireland, 2011, pp. 1723–1730. DOI: 10.1145/2001576.2001808.

[145] Sean Luke et al. "MASON: A Multiagent Simulation Environment". In: *Simulation* 81.7 (2005), pp. 517–527. DOI: 10.1177/0037549705058073.

[146] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Ed. by Yassine Lakhnech and Sergio Yovine. Berlin, Heidelberg: Springer, 2004, pp. 152–166. ISBN: 978-3-540-30206-3. DOI: 10.1007/978-3-540-30206-3_12.

[147] Rahuman S. Malik-Sheriff et al. "BioModels—15 years of sharing computational models in life science". In: *Nucleic Acids Research* (Nov. 2019). DOI: 10.1093/nar/gkz1055.

[148] Nicholas D. Matsakis and Felix S. Klock. "The Rust Language". In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology - HILT '14*. ACM Press, 2014. DOI: 10.1145/2663171.2663188.

[149] Carsten Maus, Stefan Rybacki, and Adelinde M. Uhrmacher. "Rule-based multi-level modeling of cell biological systems". In: *BMC Systems Biology* 5.1 (Oct. 17, 2011), p. 166. ISSN: 1752-0509. DOI: 10.1186/1752-0509-5-166.

[150] Orianne Mazemondet et al. "Elucidating the Sources of β-catenin Dynamics in Human Neural Progenitor Cells". In: *PLOS ONE* 7.8 (2012). DOI: 10.1371/journal.pone.0042792.

[151] Conor McBride and Ross Paterson. "Applicative programming with effects". In: *Journal of Functional Programming* 18.01 (May 2007). DOI: 10.1017/s0956796807006326.

[152] Timothy McPhillips et al. "YesWorkflow: A User-Oriented, Language-Independent Tool for Recovering Workflow Information from Scripts". In: *International Journal of Digital Curation* 10.1 (May 2015), pp. 298–313. DOI: 10.2218/ijdc.v10i1.370.

[153] José Meseguer. "Why Formal Modeling Language Semantics Matters". In: *International Conference on Model-Driven Engineering Languages and Systems, MODELS*. 2014. URL: https://models2014.webs.upv.es/speakernotes.htm.

[154] Tom Meyer et al. "On Performance Benefits of Code Generation at Runtime for Interpreted Domain-Specific Modeling Languages". In: *Proceedings of the 2018 Winter Simulation Conference*. IEEE, 2018. DOI: 10.1109/WSC.2018.8632545.

[155] John A. Miller, Jun Han, and Maria Hybinette. "Using Domain Specific Language for Modeling and Simulation: Scalation As a Case Study". In: *Proceedings of the 2010 Winter Simulation Conference*. WSC '10. Baltimore, Maryland: IEEE, 2010. ISBN: 978-1-4244-9864-2. DOI: 10.1109/wsc.2010.5679113.

[156] John A. Miller et al. "Investigating Ontologies for Simulation Modeling". In: *Proceedings of the 37th Annual Symposium on Simulation*. ANSS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 55–. ISBN: 0-7695-2110-X. DOI: 10.1109/simsym.2004.1299465.

[157] K. Jarrod Millman and Fernando Pérez. "Developing open-source scientific practice". In: *Implementing Reproducible Research* 149 (2014). DOI: 10.1201/9781315373461-6.

[158] Robin Milner. "The Polyadic π-calculus: a Tutorial". In: *Logic and Algebra of Specification*. Ed. by Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg. Springer Berlin Heidelberg, 1993, pp. 203–246. DOI: 10.1007/978-3-642-58041-3_6.

[159] Robin Milner, Joachim Parrow, and David Walker. "A Calculus of Mobile Processes, I". In: *Inf. Comput.* 100.1 (Sept. 1992), pp. 1–40. DOI: 10.1016/0890-5401(92)90008-4.

[160] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0262631814.

[161] Robert N. Moll, Michael A. Arbib, and A. J. Kfoury. *An Introduction to Formal Language Theory*. 1st ed. Springer Publishing Company, Incorporated, 2012. ISBN: 1461395976.

[162] *MPS: Domain-Specific Language Creator by JetBrains*. 2018. URL: https://www.jetbrains.com/mps/ (visited on 10/15/2018).

[163] Asher Mullard. "Reliability of 'new drug target'claims called into question". In: *Nature Reviews Drug Discovery* 10.9 (Aug. 2011), pp. 643–644. DOI: 10.1038/nrd3545.

[164] Alexandre Muzy and David R. C. Hill. "What is new with the activity world view in modeling and simulation? using activity as a unifying guide for modeling and simulation". In: *Proceedings of the 2011 Winter Simulation Conference*. IEEE, 2011. DOI: 10.1109/wsc.2011.6147991.

[165] Alexandre Muzy et al. "Refounding of the activity concept? Towards a federative paradigm for modeling and simulation". In: *SIMULATION* 89.2 (Oct. 2012), pp. 156–177. DOI: 10.1177/0037549712457852.

[166] *New_ C_ Parser - GCC Wiki*. 2018. URL: https://gcc.gnu.org/wiki/New_C_Parser (visited on 11/01/2018).

[167] Tobias Nipkow and Gerwin Klein. *Concrete Semantics*. Springer International Publishing, 2014. DOI: 10.1007/978-3-319-10542-0.

[168] Michael J. North et al. "Complex Adaptive Systems Modeling with Repast Simphony". In: *Complex Adaptive Systems Modeling* 1.1 (2013), pp. 1–26. DOI: 10.1186/2194-3206-1-3.

[169] Bryan O'Sullivan, Donald Bruce Stewart, and John Goerzen. *Real World Haskell*. O'Reilly UK Ltd., Dec. 1, 2008. 710 pp. ISBN: 0596514980.

[170] Martin Odersky and Tiark Rompf. "Unifying functional and object-oriented programming with Scala". In: *Communications of the ACM* 57.4 (Apr. 2014), pp. 76–86. DOI: 10.1145/2591013.

[171] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: Updated for Scala 2.12*. 3rd ed. USA: Artima Incorporation, 2016. ISBN: 0981531687.

[172] Martin Odersky and Matthias Zenger. "Scalable component abstractions". In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '05*. ACM Press, 2005. DOI: 10.1145/1094811.1094815.

[173] Bruno C. d. S. Oliveira and Andres Löh. "Abstract syntax graphs for domain specific languages". In: *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation - PEPM '13*. ACM Press, 2013. DOI: 10.1145/2426890.2426909.

[174] Oleksandr Ostrenko et al. "pSSAlib: The Partial-propensity Stochastic Chemical Network Simulator". In: *PLOS Computational Biology* 13.12 (2017), pp. 1–15. DOI: 10.1371/journal.pcbi.1005865.

[175] Nicolas Oury and Gordon D. Plotkin. "Coloured stochastic multilevel multiset rewriting". In: *Proceedings of the 9th International Conference on Computational Methods in Systems Biology - CMSB '11*. ACM Press, 2011. DOI: 10.1145/2037509.2037534.

[176] Nicolas Oury and Gordon D. Plotkin. "Multi-level modelling via stochastic multi-level multiset rewriting". In: *Mathematical Structures in Computer Science* 23.2 (2013), pp. 471–503. DOI: 10.1017/s0960129512000199.

[177] Jonathan Ozik et al. "The ReLogo agent-based modeling language". In: *Proceedings of the 2013 Winter Simulation Conference*. IEEE, 2013. DOI: 10.1109/wsc.2013.6721539.

[178] Ernest H. Page. "Simulation Modeling Methodology: Principles and Etiology of Decision Support". PhD thesis. Virginia Polytechnic Institute and State University, 1994. URL: http://www.thesimguy.com/articles/simModMeth.pdf.

[179] Jon Parker and Joshua M. Epstein. “A Distributed Platform for Global-Scale Agent-Based Models of Disease Transmission”. In: *ACM Transactions on Modeling and Computer Simulation* 22.1 (2011), 2:1–2:25. DOI: 10.1145/2043635.2043637.

[180] Terence Parr. *The Definitive ANTLR 4 Reference*. O’Reilly UK Ltd., Dec. 11, 2015. 328 pp. ISBN: 1934356999.

[181] Vaclav Pech, Alex Shatalin, and Markus Voelter. “JetBrains MPS as a tool for extending Java”. In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages, and Tools - PPPJ ’13*. ACM Press, 2013. DOI: 10.1145/2500828.2500846.

[182] Michael Pedersen, Andrew Phillips, and Gordon D. Plotkin. “A High-Level Language for Rule-Based Modelling”. In: *PLOS ONE* 10.6 (June 2015), pp. 1–26. DOI: 10.1371/journal.pone.0114296.

[183] Danhua Peng et al. “Reusing simulation experiment specifications in developing models by successive composition — a case study of the Wnt/β-catenin signaling pathway”. In: *SIMULATION* 93.8 (Apr. 2017), pp. 659–677. DOI: 10.1177/0037549717704314.

[184] Danhua Peng et al. “Reusing simulation experiment specifications to support developing models by successive extension”. In: *Simulation Modelling Practice and Theory* 68 (Nov. 2016), pp. 33–53. DOI: 10.1016/j.simpat.2016.07.006.

[185] Alan J. Perlis. “Epigrams on Programming”. In: *ACM SIGPLAN Notices* 17.9 (1982), pp. 7–13. DOI: 10.1145/947955.1083808.

[186] Marian Petre. “Why Looking Isn’t Always Seeing: Readership Skills and Graphical Programming”. In: *Communications of the ACM* 38.6 (June 1995), pp. 33–44. DOI: 10.1145/203241.203251.

[187] Stanford Encyclopedia of Philosophy. *Scientific Method*. Nov. 13, 2015. URL: https://plato.stanford.edu/entries/scientific-method/.

[188] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0-262-66071-7.

[189] Benjamin C. Pierce. *Types and programming languages*. Cambridge, Mass.: MIT Press, 2002.

[190] Maria E. Pierce et al. “Developing and validating a multi-level ecological model of eastern Baltic cod (Gadus morhua) in the Bornholm Basin – A case for domain-specific languages”. In: *Ecological Modelling* 361 (Oct. 2017), pp. 49–65. DOI: 10.1016/j.ecolmodel.2017.07.012.

[191] Hans E. Plesser. “Reproducibility vs. Replicability: A Brief History of a Confused Terminology”. In: *Frontiers in Neuroinformatics* 11 (Jan. 2018). DOI: 10.3389/fninf.2017.00076.

[192] Corrado Priami. “Stochastic π-calculus”. In: *The Computer Journal* 38.7 (July 1995), pp. 578–589. DOI: 10.1093/comjnl/38.7.578.

[193] Steven F. Railsback, Steven L. Lytinen, and Stephen K. Jackson. "Agent-Based Simulation Platforms: Review and Development Recommendations". In: *Simulation* 82.9 (2006), pp. 609–623. DOI: 10.1177/0037549706073695.

[194] Rajesh Ramaswamy, Nélido González-Segredo, and Ivo F. Sbalzarini. "A new class of highly efficient exact stochastic simulation algorithms for chemical reaction networks". In: *The Journal of Chemical Physics* 130.24 (June 2009), p. 244104. DOI: 10.1063/1.3154624.

[195] Eric S. Raymond. *The art of UNIX programming: with contributions from thirteen UNIX pioneers, including its inventor, Ken Thompson.* Addison-Wesley professional computing series. Boston: Addison-Wesley, 2008. 525 pp. ISBN: 978-0-13-142901-7.

[196] Oliver Reinhardt and Adelinde M. Uhrmacher. "An Efficient Simulation Algorithm for Continuous-Time Agent-Based Linked Lives Models". In: *Proceedings of the 50th Annual Simulation Symposium.* ANSS '17. Virginia Beach, Virginia: Society for Computer Simulation International, 2017. DOI: 10.5555/3106388.3106397.

[197] Oliver Reinhardt et al. "Valid and Reproducible Simulation Studies—Making It Explicit". In: *Simulation Foundations, Methods and Applications.* Ed. by Claus Beisbart and Nicole J. Saam. Springer International Publishing, 2019, pp. 607–627. DOI: 10.1007/978-3-319-70766-2_25.

[198] Tiark Rompf and Martin Odersky. "Lightweight modular staging". In: *Communications of the ACM* 55.6 (June 2012), p. 121. DOI: 10.1145/2184319.2184345.

[199] Andreas Ruscheinski, Tom Warnke, and Adelinde M. Uhrmacher. "Artifact-based Workflows for Supporting Simulation Studies". In: *IEEE Transactions on Knowledge and Data Engineering* (2019), pp. 1–1. DOI: 10.1109/tkde.2019.2899840.

[200] Andreas Ruscheinski et al. "Generating Simulation Experiments Based on Model Documentations and Templates". In: *Proceedings of the 2018 Winter Simulation Conference.* IEEE, 2018. DOI: 10.1109/wsc.2018.8632515.

[201] Stefan Rybacki et al. "Template and Frame Based Experiment Workflows in Modeling and Simulation Software with WORMS". In: *Proceedings of the 2012 IEEE Eighth World Congress on Services.* IEEE, June 2012. DOI: 10.1109/services.2012.22.

[202] Susan M. Sanchez. "Work Smarter, Not Harder: Guidelines for Designing Simulation Experiments". In: *Proceedings of the 2005 Winter Simulation Conference.* Piscataway, New Jersey: IEEE, 2005. DOI: 10.1109/wsc.2007.4419591.

[203] Robert G. Sargent. "Verification and validation of simulation models". In: *Proceedings of the 2010 Winter Simulation Conference.* IEEE, 2010. DOI: 10.1109/wsc.2010.5679166.

[204] Hans Georg Schaathun. "Evaluation of splittable pseudo-random generators". In: *Journal of Functional Programming* 25 (2015). DOI: 10.1017/s095679681500012x.

[205] Martin Scharm. "Improving Reproducibility and Reuse of Modelling Results in the Life Sciences". PhD thesis. 2018. DOI: 10.18453/ROSDOK_ID00002315.

[206] Christian Schmitt et al. "An Evaluation of Domain-Specific Language Technologies for Code Generation". In: *Proceedings of the 2014 14th International Conference on Computational Science and Its Applications*. IEEE, June 2014. DOI: 10.1109/iccsa.2014.16.

[207] John A. P. Sekar and James R. Faeder. "An introduction to rule-based modeling of immune receptor signaling". In: *Systems Immunology*. Ed. by Jayajit Das and Ciriyam Jayaprakash. CRC Press, Sept. 2018, pp. 71–90. DOI: 10.1201/9781315119847-5.

[208] Koushik Sen, Mahesh Viswanathan, and Gul Agha. "On Statistical Model Checking of Stochastic Systems". In: *Computer Aided Verification*. Ed. by Kousha Etessami and Sriram K. Rajamani. Berlin, Heidelberg: Springer, 2005, pp. 266–280. DOI: 10.1007/11513988_26.

[209] Alejandro Serrano and Jurriaan Hage. "Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules". In: *Programming Languages and Systems*. Ed. by Peter Thiemann. Springer Berlin Heidelberg, 2016, pp. 672–698. DOI: 10.1007/978-3-662-49498-1_26.

[210] Colin J. R. Sheppard and Steven F. Railsback. *Time Extension for NetLogo (Version 1.2)*. 2015. URL: https://github.com/colinsheppard/time (visited on 07/2016).

[211] Bhargav Shivkumar, Jeffrey Murphy, and Lukasz Ziarek. "RTMLton: An SML Runtime for Real-Time Systems". In: *Practical Aspects of Declarative Languages*. Ed. by Ekaterina Komendantskaya and Yanhong Annie Liu. Cham: Springer International Publishing, 2020, pp. 113–130. ISBN: 978-3-030-39197-3. DOI: 10.1007/978-3-030-39197-3_8.

[212] Gregory A. Silver, Lee W. Lacy, and John A. Miller. "Ontology Based Representations of Simulation Models Following the Process Interaction World View". In: *Proceedings of the 2006 Winter Simulation Conference*. WSC '06. Monterey, California: IEEE, 2006. ISBN: 1424405017. DOI: 10.1109/wsc.2006.323208.

[213] Eric Silverman et al. "Semi-Artificial Models of Populations: Connecting Demography with Agent-Based Modelling". In: *Advances in Computational Social Science*. Ed. by Shu-Heng Chen et al. Vol. 11. Agent-Based Social Systems. Springer Japan, 2014, pp. 177–189. DOI: 10.1007/978-4-431-54847-8_12.

[214] Eric Silverman et al. "When demography met social simulation: a tale of two modelling approaches". In: *Journal of Artificial Societies and Social Simulation* 16.4 (Oct. 2013), p. 9. DOI: 10.18564/jasss.2327.

[215] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201656973.

[216] Adam M. Smith et al. “RuleBender: integrated modeling, simulation and visualization for rule-based intracellular biochemistry”. In: *BMC Bioinformatics* 13.S8 (May 2012). DOI: 10.1186/1471-2105-13-s8-s3.

[217] Lucian P. Smith et al. “Antimony: a modular model definition language”. In: *Bioinformatics* 25.18 (2009), pp. 2452–2454. DOI: 10.1093/bioinformatics/btp401.

[218] Michael W. Sneddon, James R. Faeder, and Thierry Emonet. “Efficient Modeling, Simulation and Coarse-Graining of Biological Complexity With NFsim”. In: *Nature Methods* 8.2 (2011), pp. 177–183. DOI: 10.1038/nmeth.1546.

[219] *Standard ML*. 2018. URL: http://www.dcs.ed.ac.uk/home/mlj/doc/aboutsml.html (visited on 10/09/2018).

[220] Melanie I. Stefan et al. “Multi-state Modeling of Biomolecules”. In: *PLOS Computational Biology* 10.9 (Sept. 2014), pp. 1–9. DOI: 10.1371/journal.pcbi.1003844.

[221] Ryan Suderman et al. “Generalizing Gillespie’s Direct Method to Enable Network-Free Simulations”. In: *Bulletin of Mathematical Biology* 81.8 (Mar. 2018), pp. 2822–2848. DOI: 10.1007/s11538-018-0418-2.

[222] Josef Svenningsson and Emil Axelsson. “Combining Deep and Shallow Embedding for EDSL”. In: *Trends in Functional Programming*. Ed. by Hans-Wolfgang Loidl and Ricardo Peña. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 21–36. ISBN: 978-3-642-40447-4. DOI: 10.1007/978-3-642-40447-4_2.

[223] Tim Sweda. *Discrete Event Simulation Using Repast Java: A DiscreteEventSim Tutorial*. 2011. URL: https://code.google.com/archive/p/repast-demos/wikis/DiscreteEventSim.wiki (visited on 07/2016).

[224] Wouter Swierstra. “Data types à la carte”. In: *Journal of Functional Programming* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758.

[225] Eugene Syriani et al. “AToMPM: A web-based modeling environment”. In: *Joint proceedings of MODELS’13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*. 2013, pp. 21–25. URL: http://models2013.lcc.uma.es/pre/demos/syriani.pdf.

[226] Apostolos Syropoulos. “Mathematics of Multisets”. In: *Multiset Processing*. Ed. by Cristian S. Calude et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 347–358. ISBN: 978-3-540-45523-3. DOI: 10.1007/3-540-45523-x_17.

[227] Yentl Van Tendeloo and Hans Vangheluwe. “Discrete Event System Specification Modeling And Simulation”. In: *Proceedings of the 2018 Winter Simulation Conference*. IEEE, 2018. DOI: 10.1109/wsc.2018.8632372.

[228] R. D. Tennent. *Semantics of Programming Languages*. Pearson Education, July 1, 1991. ISBN: 0138055998.

[229] Georgios Theodoropoulos and Brian Logan. “A Framework for the Distributed Simulation of Agent-Based Systems”. In: *Proceedings of the 13th European Simulation Multiconference (ESM’99)*. Citeseer, 1999, pp. 58–65. URL: http://www.cs.nott.ac.uk/~pszbsl/papers/Theodoropoulos%20Logan:99a.pdf.

[230] John J. Tyson. “Modeling the cell division cycle: cdc2 and cyclin interactions.” In: *Proceedings of the National Academy of Sciences* 88.16 (Aug. 1991), pp. 7328–7332. DOI: 10.1073/pnas.88.16.7328.

[231] Adelinde M. Uhrmacher et al. “Panel — Reproducible research in discrete event simulation — A must or rather a maybe?” In: *Proceedings of the 2016 Winter Simulation Conference*. IEEE, 2016. DOI: 10.1109/wsc.2016.7822185.

[232] Adelinde M. Uhrmacher et al. “Reproducible Research in Discrete Event Simulation: A Must or Rather a Maybe?” In: *Proceedings of the 2016 Winter Simulation Conference*. WSC ’16. Arlington, Virginia: IEEE, 2016. ISBN: 9781509044849. DOI: 10.5555/3042094.3042264.

[233] Yentl Van Tendeloo and Hans Vangheluwe. “The Modular Architecture of the Python(P)DEVS Simulation Kernel: Work in Progress Paper”. In: *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*. DEVS ’14. Tampa, Florida: Society for Computer Simulation International, 2014. DOI: 10.5555/2665008.2665022.

[234] Philip Wadler. “How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages”. In: *Functional Programming Languages and Computer Architecture*. Ed. by Jean-Pierre Jouannaud. Springer Berlin Heidelberg, 1985, pp. 113–128. DOI: 10.1007/3-540-15975-4_33.

[235] Philip Wadler. *The Expression Problem*. 1998. URL: http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

[236] Dagmar Waltemath et al. “Minimum Information About a Simulation Experiment (MIASE)”. In: *PLoS Computational Biology* 7.4 (Apr. 2011). Ed. by Philip E. Bourne, e1001122. DOI: 10.1371/journal.pcbi.1001122.

[237] Dagmar Waltemath et al. “Reproducible computational biology experiments with SED-ML - The Simulation Experiment Description Markup Language”. In: *BMC Systems Biology* 5.1 (Dec. 15, 2011), p. 198. ISSN: 1752-0509. DOI: 10.1186/1752-0509-5-198.

[238] Tom Warnke, Tobias Helms, and Adelinde M. Uhrmacher. “Reproducible and flexible simulation experiments with ML-Rules and SESSL”. In: *Bioinformatics* 34.8 (Nov. 2017). Ed. by Bonnie Berger, pp. 1424–1427. DOI: 10.1093/bioinformatics/btx741.

[239] Tom Warnke, Tobias Helms, and Adelinde M. Uhrmacher. “Syntax and Semantics of a Multi-Level Modeling Language”. In: *Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. New York: ACM, 2015, pp. 133–144. DOI: 10.1145/2769458.2769467.

[240] Tom Warnke, Frank Krüger, and Adelinde M. Uhrmacher. “Open Simulation Software - Development and Application”. In: *ASIM 2018 - 24. Symposium Simulationstechnik*. ARGESIM Verlag Wien, 2018, pp. 307–312. DOI: 10.11128/arep.56.

[241] Tom Warnke, Oliver Reinhardt, and Adelinde M. Uhrmacher. “Population-based CTMCS and agent-based models”. In: *Proceedings of the 2016 Winter Simulation Conference*. IEEE, 2016. DOI: 10.1109/wsc.2016.7822181.

[242] Tom Warnke and Adelinde M. Uhrmacher. “Complex Simulation Experiments made easy”. In: *Proceedings of the 2018 Winter Simulation Conference*. IEEE, 2018. DOI: 10.1109/wsc.2018.8632429.

[243] Tom Warnke and Adelinde M. Uhrmacher. “Reproducible parallel simulation experiments via pure functional programming”. In: *Proceedings of the 2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, Oct. 2019. DOI: 10.1109/ds-rt47707.2019.8958655.

[244] Tom Warnke et al. “ML3: A Language for Compact Modeling of Linked Lives in Computational Demography”. In: *Proceedings of the 2015 Winter Simulation Conference*. Piscataway, New Jersey: IEEE, 2015. DOI: 10.1109/wsc.2015.7408382.

[245] Tom Warnke et al. “Modelling and simulating decision processes of linked lives: An approach based on concurrent processes and stochastic race”. In: *Population Studies* 71.sup1 (Oct. 2017), pp. 69–83. DOI: 10.1080/00324728.2017.1380960.

[246] Tom Warnke et al. “Simulation Galleries - Combining Experiment Design and Visual Analytics”. In: *INFORMS Simulation Society 2017 Research Workshop (I-Sim 2017)*. 2017. URL: http://eprints.mosi.informatik.uni-rostock.de/457/.

[247] Gezhi Weng, Upinder S. Bhalla, and Ravi Iyengar. “Complexity in biological signaling systems”. In: *Science* 284.5411 (1999), pp. 92–96. DOI: 10.1126/science.284.5411.92.

[248] Ward Whitt. *Continuous-time Markov Chains*. Dec. 2013. URL: http://www.columbia.edu/~ww2040/6711F13/CTMCnotes120413.pdf (visited on 05/16/2019).

[249] *Why Haskell just works*. 2019. URL: https://wiki.haskell.org/Why_Haskell_just_works (visited on 07/11/2019).

[250] Norman J. Wildberger. *A new look at multisets*. 2003. URL: https://web.maths.unsw.edu.au/~norman/papers/NewMultisets5.pdf.

[251] Uri Wilensky. *NetLogo*. 1999. URL: https://ccl.northwestern.edu/netlogo/.

[252] Uri Wilensky. *NetLogo 6.0.3 User Manual: BehaviorSpace Guide*. 2018. URL: https://ccl.northwestern.edu/netlogo/docs/behaviorspace.html (visited on 04/26/2018).

[253] Frans Willekens. “Continuous-Time Microsimulation in Longitudinal Analysis”. In: *New Frontiers in Microsimulation Modelling*. Ed. by A. Zaidi, A. Harding, and P. Williamson. Ashgate, 2009, pp. 413–436. DOI: 10.4324/9781315248066-16.

[254] *Xtext - Language Engineering Made Easy!* 2018. URL: https://www.eclipse.org/Xtext/ (visited on 10/15/2018).

[255] Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. *Theory of modeling and simulation: Discrete event & iterative system computational foundations*. 3rd ed. USA: Academic Press, Inc., 2018. ISBN: 0-12-813370-8.

[256] Roberto Zunino et al. "ℓ: An Imperative DSL to Stochastically Simulate Biological Systems". In: *Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*. Cham: Springer International Publishing, 2015, pp. 354–374. ISBN: 978-3-319-25527-9. DOI: 10.1007/978-3-319-25527-9_23.