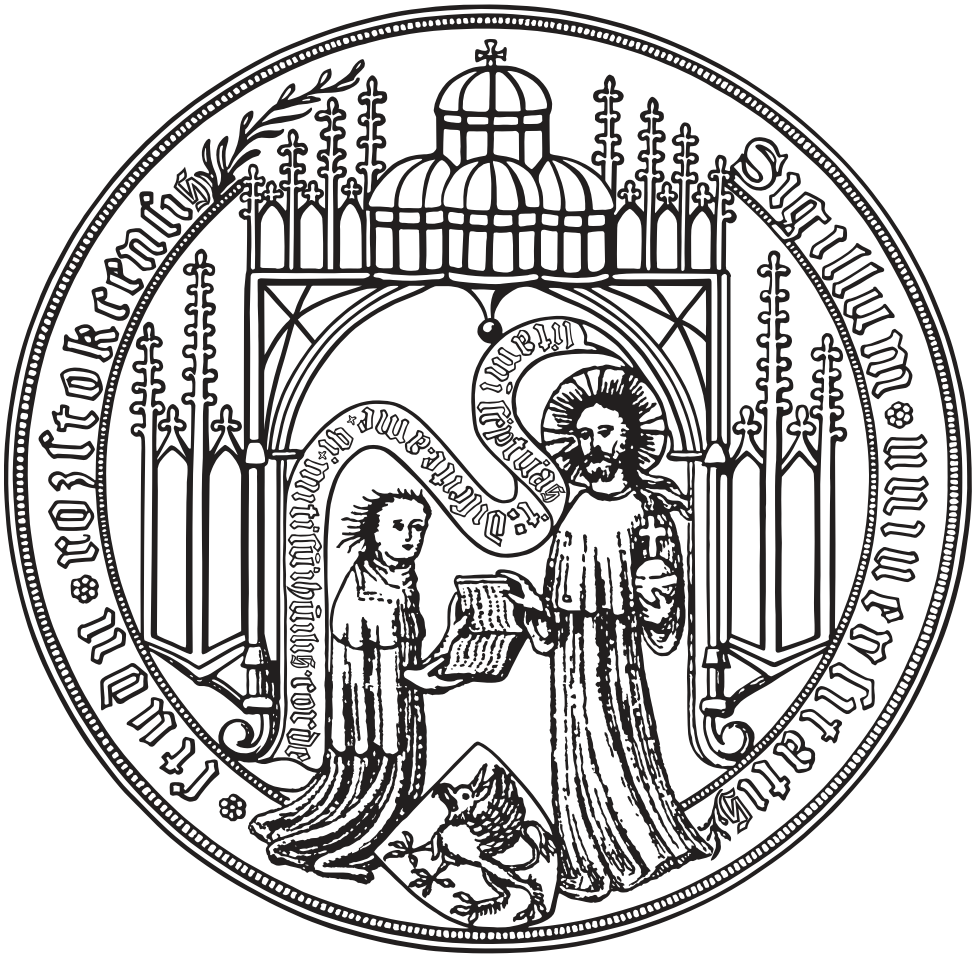


Efficient Abstraction and Execution of Stochastic Simulation Models

Dissertation
submitted for the academic degree of

Doktoringenieur (Dr.-Ing.)

Faculty of Computer Science and Electrical Engineering,
University of Rostock, Germany

submitted by

Till Köster,
born on December 14th 1993 in Flensburg

Rostock, June 30th 2024

https://doi.org/10.18453/rosdok_id00005121

Abstract

Stochasticity is found throughout cell-biological models when describing phenomena such as diffusion and the resulting random interactions of entities. This thesis is concerned with the efficient execution of these models. We cover the fundamentals of modeling from the bottom up. Reaction systems are a main topic, particularly those that use Stochastic Simulation Algorithms. We discuss efficient variations of these algorithms, including their parallel execution on CPUs and GPUs. Furthermore, the limitations of static reaction systems will be overcome by introducing dynamic compartmentalization, which is vital for many processes. The arising challenges in efficient execution (including of a spatial particle variant) are addressed through novel algorithms and implementations. Another focus will also be the technical realization of the simulation abstraction through domain-specific languages and their interplay with simulator efficiency. This includes an application of the cell-biological simulation methods to agent-based models. Finally, we consider the issues of replication and evaluation of simulator implementations, in particular, as they relate to the questions of stochasticity and determinism. These introductions lay the groundwork for the six publications that comprise this thesis's core. The articles present aspects of this thesis in more detail.

Zusammenfassung

Zellbiologische Modelle verwenden häufig stochastische Prozesse, um zum Beispiel Phänomene wie Diffusion und das daraus folgende zufällige Interagieren von Partikeln darzustellen. In dieser Arbeit behandeln wir die effiziente Beschreibung und Ausführung dieser Modelle. Insbesondere beschäftigen uns Reaktionssysteme, die von den Gillespie-Algorithmen simuliert werden. Wir beschäftigen uns hier mit effizienten Varianten dieser Algorithmen, einschließlich ihrer parallelen Ausführung auf CPUs und Grafikkarten. Einige methodische Einschränkungen dieser Reaktionssysteme werden durch das Einführen von dynamischer Kompartimentierung überwunden. Diese Kompartimentierung ist essentiell für viele zellbiologische Prozesse. Sie birgt aber auch Herausforderungen hinsichtlich ihrer effizienten Ausführung. Diese werden von uns (auch für eine räumliche Variante) durch neue Algorithmen und Implementierungen adressiert. Eng verwandt ist die Thematik der technischen Umsetzung domänenspezifischer Sprachen und ihr Einfluss auf effiziente Simulationen. Hier erweitern wir das Feld der behandelten Anwendungen auch auf Systeme interagierender Agenten. Zuletzt werden wir uns mit Problemen der Replikation und Evaluation der Implementierungen von Simulationsalgorithmen beschäftigen, auch in Bezug auf Stochastizität und Determinismus. Der erste Teil der Arbeit gibt allgemeinen Kontext zum behandelten Thema. Darauf folgen sechs Aufsätze, die den Kern dieser Arbeit bilden. Diese Aufsätze betrachten verschiedene Aspekte des Themas im Detail.

Reviewer

Prof. Dr. Adelinde Uhrmacher, Universität Rostock

Prof. Dr. Alessandro Pellegrini, University of Rome "Tor Vergata"

Prof. Dr. Kevin Burrage, Queensland University of Technology

This cumulative thesis was defended on September 9th, 2024.

Context and Summary

Core Publications

Publications

Below is a list of all publications by the author. The publications that make up the core of the cumulative thesis are boxed. They will be referred to by letters instead of numbers throughout this thesis.

Publications as First Author

Journal

[A] **T. Köster**, T. Warnke, and A. M. Uhrmacher, *Generating Fast Specialized Simulators for Stochastic Reaction Networks via Partial Evaluation*, ACM Transactions on Modeling and Computer Simulation (TOMACS) 32, 1 (2022)

Extended version of Conference paper :

[1] **T. Köster**, T. Warnke, and A. M. Uhrmacher, *Partial Evaluation via Code Generation for Static Stochastic Reaction Network Models*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (PADS 2020) (ACM, New York, USA, 2020), pp. 159–170

T. Köster devised and implemented the method, conducted the analysis, and co-wrote the manuscript.

[B] **T. Köster**, P. Henning, and A. M. Uhrmacher, *Potential Based, Spatial Simulation of Dynamically Nested Particles*, BMC Bioinformatics 20, (2019)

T. Köster devised and implemented the method, contributed to the analysis, and co-wrote the manuscript.

[E] **T. Köster**, P. Henning, T. Warnke, and A. Uhrmacher, *Expressive Rule-Based Modeling and Fast Simulation for Dynamic Compartments*, Plos One 19, 312813 (2024)

T. Köster devised and implemented the method, contributed to the analysis, and co-wrote the manuscript.

[F] **T. Köster**, O. Reinhardt, M. Hinsch, J. Bijak, and A. M. Uhrmacher, *A Fast Embedded Language for Continuous-Time Agent-Based Simulation*, Journal of Ar- tificial Societies and Social Simulation 27, 10 (2024)

T. Köster devised and implemented the method, conducted the analysis, and co-wrote the manuscript.

Conference Papers

[C] **T. Köster**, P. J. Giabbanelli, and A. M. Uhrmacher, *Performance and Soundness of Simulation: A Case Study Based on a Cellular Automaton for in-Body Spread of HIV*, in *Winter Simulation Conference* (WSC 2020) (IEEE Computer Society, 2020), pp. 2281–2292

T. Köster devised and implemented the method, conducted the analysis, and co-wrote the manuscript.

[D] **T. Köster**, L. Herrmann, P. Andelfinger, and A. M. Uhrmacher, *GPU-Accelerated Simulation Ensembles of Stochastic Reaction Networks*, in *Winter Simulation Conference (WSC 2022)* (IEEE, 2022), pp. 2570–2581

T. Köster co-devised the method, contributed to the analysis, and co-wrote the manuscript.

Conference Papers

[2] **T. Köster**, K. Perumalla, and A. M. Uhrmacher, *Efficient Simulation of Nested Hollow Sphere Intersections for Dynamically Nested Compartmental Models* in Cell Biology, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (PADS 2017) (ACM New York, NY, USA, 2017), pp. 173–183

[3] **T. Köster** and A. M. Uhrmacher, *Handling Dynamic Sets of Reactions in Stochastic Simulation Algorithms*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (PADS 2018) (ACM, New York, NY, USA, 2018), pp. 161–164

Conference Poster / Colloquium / Invited talk

[4] **T. Köster** and A. M. Uhrmacher, *Efficient Execution for Domain Specific Languages: Comparing Two Approaches for Demography and Cellular Biology*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Association for Computing Machinery (ACM), New York, NY, USA, 2023), pp. 46–47

[5] **T. Köster**, A. M. Uhrmacher, and P. Andelfinger, *Towards an Open Repository for Reproducible Performance Comparison of Parallel and Distributed Discrete-Event Simulators*, in *SIGSIM Conference on Principles of Advanced Discrete Simulation* (ACM, New York, NY, USA, 2022), pp. 31–32

[6] **T. Köster**, *Efficient Simulation of Cell-Biological Multi-Level Models*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (PADS 2019)

[7] **T. Köster**, N. M. Drüeke, and A. M. Uhrmacher, *Latency Optimized Execution of Sequential Simulators by Parallel Parameter Optimization*, in *Winter Simulation Conference* (WSC 2018)

[8] **T. Köster**, F. Hauptmann, and A. M. Uhrmacher, *Optimizing Data Structures for Highly Dynamic Content in Collective, Adaptive Systems*, in *Winter Simulation Conference* (WSC 2018)

[9] **T. Köster** and A. M. Uhrmacher, *Multi-Level Particle-Based Modeling and Simulation of Cell Biological Systems*, in *V International Conference on Particle-Based Methods* (2017)

Reproducibility

[10] **T. Köster**, *Reproducibility Report for the Paper: Workload Interference Prevention with Intelligent Routing and Flexible Job Placement on Dragonfly*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Association for Computing Machinery (ACM), New York, NY, USA, 2023), pp. 151–153

[11] **T. Köster**, *Reproducibility Report for the Paper: Spatial/temporal Locality-Based Load-Sharing in Speculative Discrete Event Simulation on Multi-Core Machines*, in *SIGSIM Conference on Principles of Advanced Discrete Simulation* (ACM, New York, NY, USA, 2022), pp. 134–137

[12] **T. Köster**, *Reproducibility Report for the Paper: Speculative Distributed Simulation of Very Large Spiking Neural Networks*, in *SIGSIM Conference on Principles of Advanced Discrete Simulation* (ACM, New York, NY, USA, 2022), pp. 141–144

reproducibility reports before 2022 were not published at PADS

Publications as Co-author

Journal

[13] P. Henning, **T. Köster**, F. Haack, K. Burrage, and A. M. Uhrmacher, *Implications of Different Membrane Compartmentalization Models in Particle-Based in Silico Studies*, Royal Society Open Science 10, 221177 (2023)

[14] F. Haack, **T. Köster**, and A. M. Uhrmacher, *Receptor/raft Ratio Is a Determinant for Lrp6 Phosphorylation and WNT/β-Catenin Signaling*, Frontiers in Cell and Developmental Biology 9, 2085 (2021)

Conference Paper

[15] P. Andelfinger, **T. Köster**, and A. M. Uhrmacher, *Zero Lookahead? Zero Problem. The Window Racer Algorithm*, in *SIGSIM Conference on Principles of Advanced Discrete Simulation* (PADS 2023) (Association for Computing Machinery (ACM), New York, NY, USA, 2023), pp. 1–11

[16] J. Li, **T. Köster**, and P. J. Giabbanelli, *Design and Evaluation of Update Schemes to Optimize Asynchronous Cellular Automata with Random or Cyclic Orders*, in *The 25th International Symposium on Distributed Simulation and Real-Time Applications* (IEEE/ACM DS-RT 2021) (IEEE Computer Society, Los Alamitos, CA, USA, 2021), pp. 1–8

[17] J. N. Kreikemeyer, **T. Köster**, A. M. Uhrmacher, and T. Warnke, *Inferring Dependency Graphs for Agent-Based Models Using Aspect-Oriented Programming*, in *Winter Simulation Conference* (WSC 2021) (IEEE Press, 2021), pp. 1–12

[18] P. J. Giabbanelli, **T. Köster**, J. A. Devita, and J. A. Kohrt, *Optimizing Discrete Simulations of the Spread of HIV-1 to Handle Billions of Cells on a Workstation*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (PADS 2020) (ACM, New York, USA, 2020), pp. 67–78

Conference Poster

[19] F. J. Müller, F. Haack, **T. Köster**, and A. M. Uhrmacher, *Computational Model of Cell Cell Interaction in Bone Remodelling*, in *ELAINE* (2024)

[20] P. Henning, **T. Köster**, and A. M. Uhrmacher, *Challenges in Reproducing an ODE Vesicle Transport Model with Particles in Continuous Space*, in *Multiscale Modeling and Simulations to Bridge Molecular and Cellular Scales* (2018)

[21] P. Henning, **T. Köster**, and A. M. Uhrmacher, *Idiosyncracies in Multi-Spatial Modeling*, in *15th Conference on Computational Methods in Systems Biology* (CMSB 2017)

1 Introduction

This chapter includes work, results, and text (copied or adapted) from the following of the author's previous works: [D].

Historical relation

From their inception, computers have been used for simulation [22]. Simulation experiments on computer systems can be executed faster than performing the algorithmic steps by hand. Speed of execution is at the core of simulation and is closely intertwined with the development of models, methods, software, and hardware. The work on underlying computer science concepts touches on issues far beyond simulation. A famous example is Simula, a programming language for simulation that introduced object-oriented programming [23]. It has had an impact far beyond its originally intended domain. Another more recent example is the field of computation on Graphical Processing Units (GPUs) [24]. Originally designed for computer graphics, the move towards General-Purpose GPU computing, which now fuels most of AI research [25], has its origins in part in Simulation [26].

What is a model?

Cellier defines a model as [27]: "A model M for a system S and an experiment E is anything to which E can be applied in order to answer questions about S." We are interested in mathematical models, as defined in *Mathematical Modeling* [28]: "Any model (including a physical model) can be defined as a simplified representation of certain aspects of a real system. A mathematical model is a model created using mathematical concepts such as functions and equations."

Stochasticity

Stochastic models are a class of models that "predicts a set of possible outcomes weighted by their likelihoods or probabilities" [29]. Stochastic modeling is a powerful means of abstracting a complex process in a compact and efficient manner [30]. However, even for these abstracted processes, the computational demands for simulation can still be very high. Beyond the execution time for a single model run, we have potentially costly parameter optimization and scanning operations. For stochastic simulation, additional replication runs are usually required to mitigate stochastic noise [31].

What is efficiency?

This thesis concerns *efficiency* which need not be congruent with other similar (and sometimes interchangeably used) terms like *fast* or *high performance*. Efficiency, as a term in simulation, entails more than just speed of execution. Efficiency, in a general sense, is the ratio of a resource's usage to the achieved output.

Time efficiency

The most obvious aspect is "time to execute an experiment." Here, we are interested in the time efficiency of our simulation tool. However, there also is the meaning of efficient use of other resources.

If the problem/model is fixed, we have limited options. One choice can be "faster" hardware. If our existing implementation is capable of making use of it, better, more modern hardware can improve execution speed. The more interesting approaches, from a computer science perspective, are optimized algorithms, data structures, and implementations.

There also is a work efficiency aspect regarding the time needed to implement the model. This is hard to evaluate, and mostly beyond the scope of this thesis.

However, some closely related questions, like the choice of helpful abstractions and ease of use, will be considered later in this thesis.

Energy efficiency

A defining resource for the financial cost of computing is electrical *energy*, or its rate of usage: *power*. There are different approaches to achieving energy efficiency [32].

By rather happy coincidence, being energy efficient is very closely related to executing in a short amount of time for fixed hardware. This concept is called *race to idle* [33,34]. The underlying principle is to keep power usage as low as possible (idle state) for as long as possible by spending a short amount of time using all available resources to complete the computation.

The ideal CPU frequency for computation, as well as the potential interplay with multiple tasks or even more power intense resources like network or storage, is subject of ongoing research [35].

Power efficient resources

Another avenue to approach (energy) efficiency is through optimized and specialized hardware and parallelization. Here, we distinguish different levels to apply parallelization [36]. In *fine-grained* parallelization, we want to make a single run of the simulation fast. On the other end of the scale, we find *coarse-grained* parallelization, where we are interested in exploiting parallel hardware to accelerate experiments with multiple replications.

Many slower CPU cores can be more power-efficient than a few faster ones due to the nonlinear scaling of frequency to power usage and other factors. This has led to a move to many core systems [37]. It was also the core idea behind the Intel Xeon Phi architecture [38], where specific accelerator hardware was added to a computer to outsource the expensive computations.

In terms of accelerators, however, the many-core approach has not stood the test of time for high-performance computing compared to another type of architecture: general-purpose computing on Graphical Processing Units (GPUs) [24]. The success of this approach to computing comes from massively parallel hardware specializing in data-parallel computation with high memory bandwidth.

General-Purpose GPUs

In the pursuit of improved efficiency, scientists have explored the usage of hardware beyond CPUs. GPUs have evolved from fixed-function hardware targeting the rendering of three-dimensional scenes to general-purpose accelerators widely available in machines ranging from laptops to supercomputers. Common GPU frameworks such as OpenCL [39] and Nvidia Cuda [40] allow developers to specify GPU programs, referred to as *kernels*, on the level of individual GPU *threads* which to a degree operate in lockstep.

The GPU architecture focuses on executing thousands of arithmetic operations in parallel rather than on control flow and caches. Therefore, algorithms involving regular control flow and memory access patterns are a natural fit for GPU-based parallelization. Although simulations often involve sparse and irregular computations and thus require GPU-specific adaptations and optimizations, simulations from various domains, including cell biology, have been shown to benefit immensely from GPU-based parallelization given suitable choices of algorithms and data structures [41–43].

Outline

This thesis will cover different aspects of efficiency. First, we will cover the domain of abstractions, specifically the use of stochasticity for abstraction and the related implications on correctness and performance in Chapter 2. Next, we will discuss in more detail the simulation of reactions in Stochastic Simulation Algorithms in Chapter 3. In Chapter 4, we will delve into the particular challenges of efficient simulation of dynamic structure models. Finally, the issues of benchmarking and replication are discussed in Chapter 5. This will prepare the reader for the six publications that make up the core of this thesis.

2 Stochasticity in Simulation

The previous chapter outlined in broad strokes the overall topic of *efficient simulation*. In this chapter, we will investigate one particular aspect, namely *stochacticity* and its interplay with efficient simulation. First, we will discuss using stochastic processes as an abstraction (Section 2.1). In this chapter, we will follow a bottom-up approach from ab initio methods to the levels of abstraction used in the later parts of this thesis, where reactions are the main object of interest (Section 2.2). Next, we will discuss a particular case study of exploiting implementation stochasticity for fast execution and the limitations inherent to this modeling approach (Section 2.3). These limitations will be partially resolved by introducing event-driven stochastic methods (Section 2.4), which will, in turn, serve as the foundations for the next chapters.

This chapter includes work, results, and text (copied or adapted) from the following of the author's previous works: [44], [C], and [B].

Introduction

In a stochastic process, a series of random values follow a specific probability distribution. In modeling and simulation, stochasticity is a very powerful abstraction [30]. It is used whenever a process is either (or a combination of) too complex or costly to simulate in detail or not understood well enough to be simulated in detail. The choices associated with stochasticity in modeling are fundamental to the abstraction process in modeling [45,46].

Queueing example

Take, for example, a simple queuing model [47] of a supermarket register. The duration it takes to process an individual custom depends on many aspects. The exact goods purchased, the detailed layout of the register, the speed at which the goods are presented at the register, etc. If those dependencies are part of the underlying research question, then they need to be modeled. However, if a broader question is of concern, a stochastic process could be assumed as part of the abstraction process inherent to modeling. For example, our data might suggest that the duration of processing follows a Poisson distribution. Here, stochastic modeling (instead of including all the potential details) has two main advantages: Firstly, it is easier to understand and reason about. Secondly, it is faster to execute.

Top-down vs. bottom-up

Furthermore, a more complex model might not even be possible due to either a lack of data or understanding. The right level of abstraction is also naturally related to the research question at the beginning of the study. A high level of detail or complexity is not necessarily helpful in this regard. Here, the objective of the research study is important. Broadly speaking, we find two approaches in modeling: *bottom-up* and *top-down* [48]. In a bottom-up approach, we build our model *up* from first principles or individual components. We take these components and merge and combine them as needed to create the model. A top-down approach, on the other hand, tries to look at the overall behavior and iteratively build a model that matches this. Starting from the top, we move down by adding and expanding components of the model. In practice, both strategies can be used, even in building the same model [49]. In this Chapter, we will find a bottom-up derivation of cellular biology methods in Section 2.1 and a discussion of the performance and problems of a top-down cellular automaton model of HIV spread in Section 2.3.

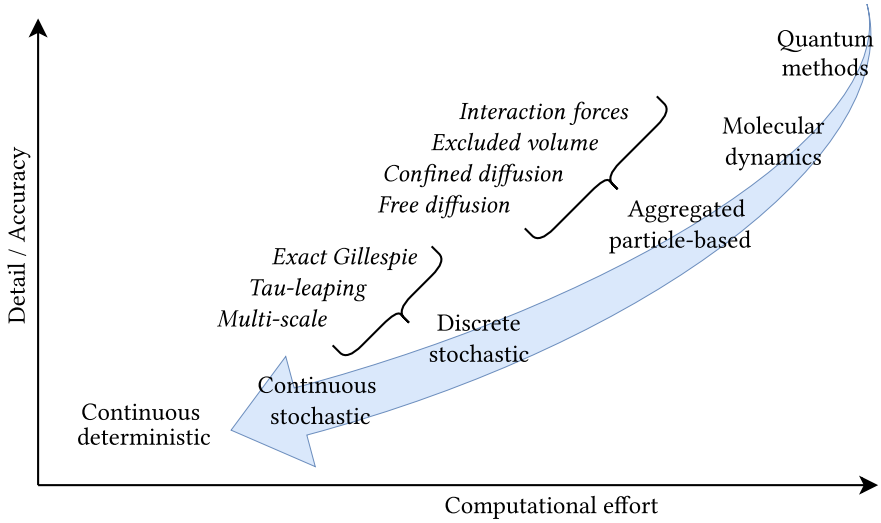


Figure 1: Overview of methods adapted and extended from [50]. The methods are placed roughly in line with their associated detail/accuracy compared to the computational cost. We observe an overall tradeoff between these two aspects. This thesis will focus on *aggregated particle-based* methods (where one particle represents a complex compound structure) with forces in Section 4.2, as well as *discrete stochastic* methods in Chapter 3. The blow arrow indicates the direction of bottom-up abstraction, which Section 2.1 follows.

2.1 Abstraction in Cellular Biology

In this next section, I will detail the process of applying stochastic abstractions to the fundamental processes to find an appropriate level of abstraction. The following paragraphs follow Figure 1 from the top right to the bottom left corner. The goal is to model and simulate biological processes like signalling [51], metabolism [52], etc., at the cellular and subcellular levels. Systems Biology has made great progress in our understanding [53] at this scale. Whole-cell simulation has been coined a "grand challenge of the 21st century" [54]. In this section, we will structure the method space to find suitable approaches for the different aspects of this grand challenge.

Starting at the bottom

Biological systems are, like all matter, made up of basic subatomic particles (electrons, neutrons, and protons). Their behavior can be modeled through the fundamental differential equations of quantum theory. The Dirac or Schrödinger equations [55] are, as far as we know, correct and fully describe the behavior of an object. These most fundamental approaches are, however, extremely computationally expensive and, even worse, scale exponentially, leading to the term of the *exponential wall* that limits computational capability, as even a significant improvement in computational power only allows for insignificantly larger systems to be solved. The current state-of-the-art methods are only able to fully capture the behavior of a molecule on the order of a few atoms at nanosecond timescales [56]. When considering the goal of simulating cellular processes, these advanced methods are not very helpful, seeing as a single macro-molecule in the cytosol of a cell, like hemoglobin, consists of ten-thousands of basic particles, meaning that solving the Schrödinger equation for a reasonably large system of interest lies magnitudes beyond the current computational power and (keeping

the exponential scaling in mind) any attempt at a full and exact solution to the problem is futile.

Need for approximation

At this point, we need to approximate the fundamental equations using a computationally tractable method. At the scale of, for example, the hemoglobin molecule mentioned above, the predominant methods are *molecular dynamics* simulations [57], where the atomic nuclei are described as classical particles propagated by equations of motion with forces calculated from their electronic structure. Like most problems in physics and chemistry, this again results in having to solve partial differential equations, like the Hamilton or Newton equation. In practice, the systems are simulated using numerical integration schemes implemented in tools like Gromacs [58] or LAMMPS [59]. Beyond the quantum chemistry methods [60], the most common interaction here is *force-field* or *molecular mechanics*, where all interaction is integrated out to potentials like Morse or Lennard-Jones. Computationally, this has a worst-case scaling of $\mathcal{O}(n^2)$, which is already more tractable than exact quantum methods. Furthermore, in practice, scaling is closer to linear due to numerous numerical cutoff schemes [61]. Molecular dynamics find use, for example, in simulating parts of the cell membrane [62] or for investigations into the structure of biomolecules [63].

Brownian motion

Most of the computational effort in these simulation systems would now be devoted to simulating the system's interaction with solvents or other environmental materials like water. As discussed above, this is an opportunity for a stochastic approximation. Instead of computing every solvent particle explicitly using elastic collisions, we can just model random pushes to happen as part of a random process. One way to describe this resulting *Brownian Motion* is using the Langevin equation [64]:

$$m\frac{d\vec{v}}{dt} = -\lambda\vec{v} + \vec{\eta}(t)$$

This equation expands on Newton's equation of motion (without external force) $m\frac{d\vec{v}}{dt} = \vec{0}$. A dampening against the direction of motion ($\lambda\vec{v}$) and a random force ($\vec{\eta}(t)$) are added. This force captures the aggregated effect of random collisions with nonreactive elements of the system well. While the details of the random force are slightly involved mathematically [65], in practice, the random displacement is sampled from a Gaussian distribution during each integration step of the differential equation.

2.2 Artificial Chemistry

By simulating entire molecules or complexes as particles instead of individual electronic configurations, we have taken away the core feature of reactivity from the model. Particles can interact only by bumping into one another but can not merge or create more complex structures other than basic potential-based binding. Therefore, we have to artificially reintroduce reactivity into the system. Conceptually, when modeling a real-world process, the modeler manually adds specific reaction features to particular classes of particles. These types of reaction rules are the building blocks for *artificial chemistries* [66]. In Chapter 3, we will discuss the abstraction of reaction systems and their use in domain-specific languages and simulators in more detail. Artificial chemistry methods can broadly

be categorized on a spectrum from fully *spatial* [67] to the more traditional *non-spatial* methods [68]. Overall, the landscape of spatial methods is complicated, with numerous tradeoffs in accuracy, speed, and expressivity [69].

Takahashi classification

Takahashi [70] categorized simulation approaches by their choice of representing space. Below, we find a slightly adapted version of this structure.

a. *Stochastic non-spatial dynamics*: Approaches from this category are very common [71]. Momenta and even positions for all particles are ignored under the assumption that the number of non-reactive collisions is far larger than the number of reactive ones [72]. The state consists exclusively of the copy numbers (i.e., populations) of the various species. Time propagation can now be described by the Chemical Master Equation [73]. The underlying stochastic process here is a *Continuous-Time Markov-Chain* [74], a jump process with exponentially distributed sojourn times. The efficient simulation of this system is largely based on works by Gillespie [75]. This abstraction is of particular importance for this thesis, and its efficient simulation will be discussed in more detail in Chapter 3. Beyond the exact stochastic methods, τ-leaping [76] and multi-scale [77] also exist, which are increasingly inexact but more efficient approaches towards sampling the underlying jump process. Continuous simulation of the stochastic processes [78,79] is also possible [80]. Here, we can no longer capture the effects of the small numbers of specific populations, as only concentrations are considered, but stochastic noise is retained in the simulation.

b. *Compartmental dynamics*: Many tools [81,82] allow constraining the dynamics of species to specific compartments. However, compartments themselves can also be subject to dynamics. For example, compartments can fuse and divide [83]. The question of simulating these (and related) dynamical structures efficiently whilst retaining a seperation of concern is discussed in Chapter 4.

c. *Individual particles* moving in continuous space: Particles can be identified by their unique position in space, bimolecular reactions are triggered by collisions of particles, and typically, particles diffuse by brownian motion [84,85].

d. *Partial differential equations*: Spatial gradients of concentrations are calculated deterministically [86].

e. *Spatial stochastic dynamics*: Multiple particles can occupy a position within a lattice space and diffuse between neighboring positions [87].

Levels of particle-based methods

Particle-based approaches (category *c*) are the subject of a further categorization suggested by Schöneberg [88]. While in molecular dynamics particles correspond to individual atoms, in most cellular biology simulation tools, particles correspond to entire molecules (See *aggregated particles* in Figure 1). Spatial particle-based simulation tools are categorized into those that support

- *level 1 – free diffusion*: basic 3D diffusion of point particles and reactions between them are considered,
- *level 2 – confined diffusion*: diffusion can be constrained to compartments,

- *level 3 – excluded volume*: point particles are replaced by volumetric entities, and
- *level 4 – potentials for particle-particle interaction*: instead of a simple rejection of movements assuming rigid cells, potentials are used to determine the interaction of particles in terms of exclusion of movement or reactions.

One example of a level two software was the original Smoldyn [85]. Here, point particles do not hamper each other's movement but are equipped with binding and unbinding radii. The motion can be constrained to compartments. A more recent version includes collision strategies to reach *level 3* [82]. SpringSaLaD [89], ReaDDy [90], and SRSim [91] are based on forces to model the particle-particle interactions (*level 4*). Still, the levels introduced do not necessarily imply that a tool that supports *level 3* provides all the interesting features that a tool working at *level 2* offers to study the system of interest. For example, Donovan [92], whose spatial simulation works at *level 2*, supports arbitrarily complex 3D mesh geometries, whereas in ML-Space [93] at *level 3* only rigid spheres are considered.

Multi-regime methods

Merging both classifications, approaches that combine different spatial representations can be characterized. The Two-Regime method [94] and Klann [95] allow the combination of particle and grid-based diffusion dynamics within one model: *c(3)-e*. Similarly, in [96], particles and partial differential equations are combined to focus on the spatial region of interest: *c(3)-d*. [97] couples a partial differential equations solver and Smoldyn *c(2)-d*. ML-Space combines particles at *level 3* (as excluded volumes are considered), compartmental dynamics, and spatial, stochastic simulation on a grid: *b-c(3)-e* [93].

In addition, simulation environments offer the possibility to select different spatial semantics for one model specification. For example, in VCELL [86], models can be interpreted by a particle-based simulator (i.e., Smoldyn) or a partial differential equation solver, both confined to realistic geometrical compartmental structures (*c(2)*, *d*). This list is far from being complete, and the characterization of the simulation tools may only depict a specific state of development.

Nesting in ML-Force

Dynamic structures, as needed by category *a)*, are the subject of Chapter 4. In particular, in Section 4.2, we will describe a novel simulation method, called ML-Force [B], that includes capabilities of *a-b-c(4)* with an emphasis on dynamic spatial nesting.

2.3 Case study: Probabilistic Cellular Automaton

The interplay between stochasticity and efficiency is not always a trivial one. In this section, we will describe a case study of one particular model of the spread of human immunodeficiency viruses (HIV) in a multicellular environment. We were able to exploit the low entropy in the stochastic processes to drastically improve the performance [C]. We found a 2.4x performance increase compared to an already optimized baseline.

This work further builds on our work done to optimize this model in its original Python realization [18,16].

Model

The model is based on research by dos Santos [98]. It describes the in-body spread of HIV. To understand the performance optimizations, only some of the

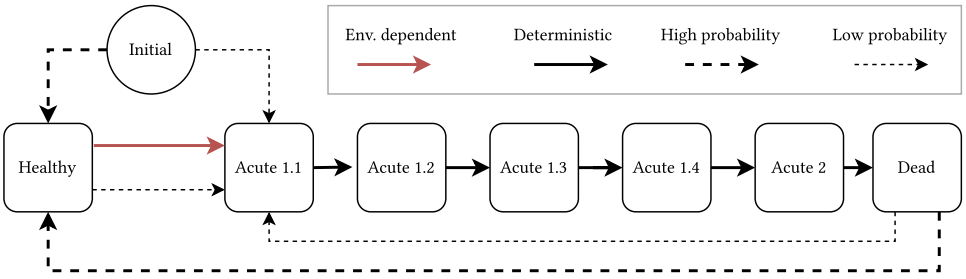


Figure 2: Flow of the dos Santos model (adapted from [C]). The 7 states are denoted by boxes, and the arrows indicate possible transitions. Most transitions are either deterministic (plain line) or have a very high or low probability (broad or narrow dashed line).

model's properties are needed as outlined in the following. The biological cells are arranged as cells in a rectangular 2D cellular automaton [99]. The system is updated globally, where each cell updates its state based on its previous state and its environment (Moore neighborhood). The seven states (corresponding to various levels of health or acuteness of the infection) and their transitions are depicted in Figure 2. Most transitions are deterministic and independent of a cell's surroundings. There are a few stochastic transitions, but these have very high/low probabilities. There is one transition that depends on a cell's neighborhood.

Baseline

First, we realized a baseline implementation of this model with the goal of getting the "fundamentals" right. That means we used a good memory layout, efficient state representation, and other best practices of performance software engineering fundamentals [100]. As a performance metric, we will use Melem/s, which is a million elements processed per second. As we will see in Chapter 5, the choice of performance metric is both important and non-trivial. It should convey information on efficiency with as few side effects as possible. In this case, giving this throughput number is helpful as it is independent of the system size. We found that such an implementation has a throughput of between 76 Melem/s and 91 Melem/s depending on the random number generator used.

Best-case experiments

We wanted to understand the limits of implementing a cellular automaton in this way, so we built simpler versions of a cellular automaton to identify the challenges to performance. In a simple, purely deterministic model with no neighborhood interactions, we get a throughput of 788 Melem/s. Interestingly, we noticed that by adding another transition with very complex rules that are never triggered, the performance of this idealized model drops to 156 Melem/s. This is very interesting and an indication that a more complex rule, even if it is never fired, limits the potential auto-vectorization of the code. We conducted further tests as outlined in the paper [C]. The conclusion is that optimal performance is achieved when transitions are uniform and, therefore, vectorizable. We have two transition types that don't meet this requirement: the stochastic transitions and the neighborhood-dependent transitions.

Optimizations

The next step is to optimize specifically for these two types of transitions. For the neighborhood calculations, we tried different optimizations, including vectorization of individual neighborhood checks. However, what performed best was to do a neighborhood check for every cell by means of vectorized addition over

the entire state representation matrix. This has some overhead when the number of neighborhood checks is small. However, since all checks can now be done at once, it improves the performance of the average neighborhood check. The second optimization is more directly related to the topic of this chapter and concerns the random transitions. Since the stochastic rates are quite extreme (i.e., very high or low) in this model, instead of doing an explicit random number check on every transition, it is mathematically equivalent to computing the total number of transitions that should have occurred or failed, respectively. We can compute this number using the binomial distribution. Now, during the simulation step, the probabilistic transitions are assumed to be deterministic, and the stochasticity is added retroactively by visiting a random subset of cells based on the binomial distribution. If the (high probability) transition has occurred in these cells, it is rolled back, and vice versa for the low probability. Overall, we see a 2.4x speed up when comparing the fastest average throughput of 182 Melem/s with the baseline of 76.4 Melem/s for high-quality random numbers.

Now, we have improved the performance of this existing model, but working with it revealed challenges in the chosen abstraction. Next, we will take a look at how the very things that could be exploited for performance gain also put into question some of the model's soundness. Another criticism investigating the original model's claims has been published as a comment to the original paper [101].

Challenges of synchronous cellular automaton

The model was designed *top-down*. There was no rigorous formal derivation of the cellular automaton approach. It was built with the goal of reproducing different phases of HIV spread based on existing data and less on underlying processes. These phases are an initial uptick in infected cells (phase one), followed by a strong reduction with low levels of infection (phase two), and a final steady state of intermediate levels of infection (phase three). However, it is well known that synchronously updated cellular automata are problematic for many real-world simulation applications [102]. Therefore, cellular automata models that simulate tumor growth use an asynchronous updating scheme instead: "These methods [cellular automata] usually update the lattice sites in a random order to reduce grid artifacts" [103]. This is in contrast with the dos Santos model, where (emphasis added) "in one time step the entire lattice is *updated in a synchronized* parallel way" [98]. If a cellular automaton with a regular grid is executed synchronously (as is the case with the dos Santos line of models), some directions are favored over others. This leads to the known problem of anisotropic behavior [104]: a direction-dependent speed of information spread throughout the system, usually favoring diagonals. In addition, fixed time steps in simulations, as typical for cellular automata execution, can introduce further errors. Therefore, fixed-time steps should usually only be used if this is inherent to the system itself. A discussion can be found in the literature, for example, in Appendix 1A of [105]. In the next section, we will consider some potential remedies for this problem.

2.4 Beyond fixed step: Discrete Event Simulation

In this section, we will examine the discrete event simulation (DES) formalism [22], which may help resolve the model's problems from the previous section and will be fundamental for the methods presented in the later chapters of this thesis.

Event Driven In discrete event simulation, the central concept is *events*. Each event has a real-numbered time stamp. Events are processed in nondecreasing timestamp order. Each event's execution may result in the scheduling of future events. As such, there is generally a very clear chain of causality between state changes. Further, the system state is well-defined at any point in time. Discrete event simulation also provides a potential avenue for parallel execution. Various methods for parallelization exist [106,107] if the state can be partitioned into separate entities that can process events locally and exchange events between them.

Tie-braking One known issue with discrete event simulation, related to the synchronicity problems of Cellular Automata, is simultaneous events and the related question of tie-braking [108]. These co-incident events can even occur in the case of fully random sampled time stamps due to the limitations of floating point representation [109], i.e., the *birthday problem* [110]. This issue has continued to receive attention, particularly in the realm of parallel discrete event simulation [111,112]. As we will see in Section 5.2, differences in tie-braking can lead to problems when deterministic execution across simulation tools is desired.

DES as a solution For our problem with the HIV model, discrete event-based simulation appears as a natural remedy to counterbalance the problems introduced due to the synchronous, step-wise computation of the cellular automata. Approaches such as Cell-DEVS and similar [113,114] exploit the potential of discrete event simulation for the cellular automata category of models. They offer the possibility to explicitly associate states with sojourn times and thus have a natural notion of continuous time and can be adapted to arbitrary networks beyond grids. We have tried to implement the dos Santos model as a discrete event simulation using exponentially distributed sojourn times as well as explicit delays. Although the conceptual translation appears straightforward, the resulting model behavior differs significantly. Instead of the initial infection reduction (second phase), we transition directly to the stable random state phase (third phase), where all model states are randomly mixed. Regular structures that shield infected cells from becoming sick do not occur when transitions are allowed in continuous time. We find that the desired three-phase behavior could only be reproduced because of the synchronous update.

Another indication that the observed behavior is the result of the synchronous, step-wise update is the question of what happens if we reduce the step size of the original stepwise model. One would expect that if the time step is reduced from one week to one day, the probabilities per step will be accordingly adapted to obtain the same results. However, the smaller probabilities per time step allow for the regular patterns of propagation to be interrupted. As a consequence, they dissolve quickly. Thus, the second phase of the observed behavior will again not occur.

How to transform? This raises the question of how to transform transition probabilities into transition rates. To my surprise, in casual conversation, many modelers assume this to be a trivial task. However, on more detailed introspection, it is neither trivial nor always possible. [115]

Timestep and rates Many models use a discrete, synchronous time step to update all entities at the same time, as in Section 2.3. Here, at each step, all model entities are updated and

transitioned based on some stochastic distribution based on their state. In fact, in some domains, time-stepped models are arguably more popular than continuous-time discrete event formulations. For example, a very popular tool in agent-based modeling is Netlogo [116]. NetLogo has time extension, which *in principle* would allow for DES-like formulation, but that is rarely used. At first glance, the discrete stepwise approach is conceptually appealing because it seems to have a lower barrier to entry. Furthermore, visualization is more straightforward. Despite their popularity, several known issues can arise if these models are not carefully designed and validated [117].

2.5 Summary

In this chapter, we have discussed stochasticity and its relation to both modeling and efficient simulation. In particular, we now have a general understanding of the methods in the field of cellular biology. We have a clearer understanding of the capabilities and limitations of the different tools in this domain. Next, we will examine how to express these formal simulation models using domain-specific languages and how to overcome the associated challenges of efficient simulation.

3 Reactions as Interfaces for Simulation

In the previous Chapter, we discussed stochasticity and its relation to modeling, simulation, and efficiency. We will address a particular class of simulation models (stochastic reaction networks) and, importantly, its interface to the modeler in this chapter. After a brief discussion of the need for interfaces, we will describe Gillespie's stochastic simulation algorithms in more detail.

First, we discuss the fundamental terminology (Section 3.1). Next, we will provide an overview of the existing algorithms for the field (Section 3.2). We also discuss the challenges and opportunities of implementing these Algorithms in parallel (Section 3.3). A particular variation used in our implementations is described (Section 3.4). Finally, we will present work based on simulator specialization that overcomes some of the previously existing overheads created by introducing a reaction interface (Section 3.5).

This chapter includes work, results, and text (copied or adapted) from the following of the author's previous works: [1], [3], [15], [A], [D], and [E].

Need for interface

When building a model of a real-world process, we need domain experts to function as model designers. It is common for these domain experts to use a general-purpose programming language and start programming. This has several disadvantages [118]: First, it is error-prone. Simulation algorithms can be hard to implement. Second, it is probably not efficient. Model designers focus on a model, not algorithm optimization. Third, it makes the models hard to reuse or repurpose.

Separation of concern

The core problem here is the lack of separation between the model and the simulator. This boundary is generally hard to draw. The key here is an appropriate interface [119]. A good simulation tool provides an abstraction that comprises a formal semantic at just the right level. Next, we will discuss different ways to realize an abstraction.

The question of a user interface is highly complex, and in this thesis, only a small subset of potential research directions is considered, with a focus on simulation performance. A plethora of in-depth discussions can be found in the literature [118,120,121,45].

Levels of abstraction

A good abstraction is a matter of seperation of concern. An abstraction is a concept used to hide underlying complexity. An interface is the technical realization of that abstraction [122]. This boundary choice between what the modeler wants to use and the model requires a lot of thought. These interfaces are on a spectrum of how close they are to either an existing programming language or a standalone tool. This does not mean that a more abstracted tool is easier to use. However, different design decisions have different implications.

Choice of domain-specific languages

In this work, I will focus on domain-specific languages [120,118], though the transition to an API is fluent. Domain-specific languages (DSLs) are languages optimized for use in a specific application. Broadly, two types of domain-specific language are distinguished. *Internal* DSLs are realized within the syntactic and semantic confines and capabilities of a host language. *External* DSLs are independent of previously existing languages. They can have an entirely unique syntax and semantics.

Great abstractions

A great example of a good abstraction is the concept of parallel discrete event simulation [106]. Writing a *functional* sequential simulator here is relatively simple and sufficient to understand the model. However, implementing an *efficient* (parallel) simulator is a matter of continued research [123,124,15]. The elegance lies in the fact that both model designers and simulator engineers have a shared abstraction to reason about.

Success of simple approaches

Having a (seemingly) simple abstraction and interface is also one of the reasons for the success of stepwise simulation tools like NetLogo [116], whose complications were discussed in Section 2.4. Even though the actual implementation is complex [125,126], NetLogo is perceived to be "simple." The remainder of this chapter concerns another very successful abstraction and its efficient realization: Gillespies's Stochastic Simulation Algorithm and its reaction abstraction.

3.1 Gillespie's abstraction

In Section 2.2, we introduced the Gillespies Stochastic Simulation Algorithm's level of abstraction [72,50]. In this section, we will discuss how an abstraction at this level can be realized by means of *reaction-based modeling*.

Reaction model

Key to this abstraction is that we don't track individuals but instead do our computations only on the total populations of classes of individuals called *species* [50]. The core concept is the *reaction*. The reactions describe how populations of certain species interact and form new species in discrete transition processes based on a formal Continuous-time Markov Chain (CTMC). Each reaction comprises *reactants* whose population decreases during the transition and *products* whose population increases during the execution. Of course, for example, during catalytic processes, a species can be both reactant and product and, as such, not be affected in its population by the reaction. Finally, a mathematical expression called the *rate function* indicates the speed of the reaction.

Propensities

Every possible reaction (or transition if viewed from a CTMC perspective) is associated with a propensity function. The propensity is the probability density of a transition occurring in the system. It depends on the current system state. In a pure CTMC, it is not time-dependent, though this formal limitation is frequently broken in practice, with tools supporting time-dependent reactions or time-triggered events [127] (e.g., in Copasi [81] or GillesPy2 [128]). The simulator usually still operates with the CTMC semantics, with some extra logic added on top to handle the special cases or even introduce delays in a more general fashion [129].

Mass action kinetics

For cellular biology, we can motivate the CTMC abstraction from the Chemical Master Equation [73]. From that rigorous point of view, the reactions are based on mass action kinetics [130]. That means that in the well-mixed abstraction, a single mass action constant related to the reaction's activation energy describes the reaction rate. The propensity is then computed by multiplying that reaction constant with the amounts of the reactants in the system. This corresponds to the higher probability of a collision occurring when the copy number increases. This is motivated by an increase in concentration for a fixed-volume system.

Arbitrary rates

For some applications, we have to go beyond mass action kinetics. Non-mass action rates usually fall into one (or both) of two categories.

- Aggregates processes, where abstracted rates are chosen for cases, where mass action rates would be possible, but a more complex rate expression reduces the number of reactions that need to explicitly occur, thereby improving performance
- Cases where the underlying modeled behavior is no longer describable within the limits of the Gillespie abstraction.

Next, we will give examples of both.

Aggregate processes

Arguably, the most famous aggregate process rate expression is the *Michaelis-Menten kinetics* [131,132]. Here, an enzyme kinetic process is aggregated for the case of low enzyme concentration. For the purpose of efficient simulation, a repeated binding and unbinding process is omitted [133]. A similar aggregate description can be made in cases of a frequent in-and-out shuttling process or other binding-unbinding reactions. The mathematically related Hill kinetics [134] has also been used for such model simplifications [135]. There is some relation between this type of model-side approach and those cases where the simulator makes a separation into slow and fast transitions for similar performance purposes [136].

Complex rate behavior

There are also cases where more complex underlying semantics beyond well-mixed particles need to be described. This is similar to the process of reintroducing the model fidelity described in Section 4.2. For example, to model oxidative phosphorylation of the mitochondrial respiratory system, electromagnetic fields bring a need for more complex rate expressions [137].

Reactions from rules

In modeling practice, not all reactions are generated by hand. Instead, a type of pattern expression called *rules* is used to generate reaction systems based on patterns. The details of this will be elaborated on in Section 4.1.

3.2 Simulating reactions

Now, how do we actually simulate these models? For this, we can have a look at Gillespie's two originally proposed algorithms [75,138], namely the *First Reaction Method* and the *Direct Method*. Both have resulted in a lineage or family of approaches.

First Reaction Method

For each transition in the system, we can compute a propensity p_i. Now, we are interested in the point in time when this transition occurs. The actual point in time when a reaction occurs follows an exponential distribution with rate $\lambda_i = \frac{1}{p_i}$. The *First Reaction Method* samples a random number from the exponential distributions for all reactions. We then take the first reaction and execute it. The execution here means that we decrease all the populations of the reactants and increment the populations of the products of the reaction. We then restart by recomputing all propensities, as they depend on the populations that might have changed.

Next Reaction Method

Dependency tracking

The *Next Reaction Method* (NRM) [139] introduced one of the most important algorithmic ideas, at least for larger SSA systems: *dependency tracking*. Instead of throwing away every computed timestep in the first reaction method, a sched-

uling approach is used. A reaction is scheduled to a specific time point, and all these time points are stored in a priority queue. The key data structure now is an added dependency graph. Each reaction stores a list of all reactions whose propensities change if that reaction fires. Therefore, only a minimal amount of propensity recalculation is needed for every step. A particularly clever idea by the authors [139] is that they do not even need to resample the random number when they update the propensity of an affected reaction. The old random number can be reused utilizing some smart but relatively simple mathematics.

The details and challenges of applying dependency tracking in dynamic systems within an internal domain-specific language are discussed further in Section 4.3.

Direct method

Another methodological approach, called the *direct method*, does not draw a random time point for every reaction in the system but uses a mathematically equivalent formulation, where the wait time until the next reaction is exponentially distributed with a $\lambda = \frac{1}{\sum_i p_i}$.

Therefore, we only need to account for the propensities and their respective sums. The most basic form of the direct method uses this but recomputes the individual propensities as well as their sum at every step.

Most proposed optimizations for SSA are built on the direct method. In fact, we're not aware of any other major advancement around the first reaction method or the next reaction method. Inspired by the next reaction method, a dependency graph is used for variants of the direct method that scale.

Phases of direct method

A typical modern variant of the direct method consists of three phases:

1. Random Number generation for timestep and weighted random choice
2. Reaction Selection
3. Reaction Execution
   - Update to the population
   - Update the propensities as needed using a dependency graph

Summation

Implementations and algorithms vary in how they select the reaction efficiently and how they compute the sum of the propensities with minimal effort. In practice, it is observed that not all transitions are equally frequent, at least for larger systems, which are interesting to optimize. Therefore, it has been proposed that reactions be sorted, making the linear weighted random choice faster. This sorting can be done ahead of time (optimized direct method [140]) or dynamically to suit changing needs at runtime (sorting direct method [141]). Another approach is to use a logarithmic search [142] instead.

The reaction sum update is also interesting. It is common practice, e.g., suggested by the optimized direct method to update only a cumulative sum. However, this could lead to numerical floating point summation errors for long-running simulations.

Advanced methods

There exist also some approaches that break with the typical patterns. Most prominent here are *partial propensity methods* [143]. Here a more complex rejection based sampling is used to minimize costly propensity updates.

Other notable approaches are approximate methods, like τ-leaping, where dependent reactions are only updated at regular (or adaptive [144]) intervals [76]. The accuracy of these methods is unbound due to the potential for negative populations [145], which can be mitigated by advanced methods [146].

3.3 Gillespie in Parallel

It is generally hard to create parallel SSA implementations that will give a good speedup compared to a fast sequential implementation. This is because the SSA system is typically highly interconnected. SSA has event rates beyond 10s of Millions per second for small systems. Even for larger systems, the even rates can be very high. The opportunities to identify potential parallel processes are limited, and advanced methods are needed [147].

Coarse- and fine-grained

The parallelization strategies for parallel SSA simulations found in the literature can be divided into two groups [36]. In coarse-grained parallelization, multiple *entire simulations* are executed in parallel. Doing this is (mostly) trivial on CPU systems, but the GPU execution model has some interesting opportunities to share work. Fine-grained parallelization, on the other hand, also parallelizes *individual simulations*, in addition to executing multiple simulations concurrently, for instance, in [148]. In the Direct Method, for example, a fine-grained approach may parallelize the updates of the population counts and propensities. The choice between coarse-grained and fine-grained parallelization determines the resources that can be dedicated to each individual simulation run. Fine-grained approaches can utilize many parallel threads to execute a single run. Hence, coarse-grained parallelization focuses on supporting particularly large numbers of replications of comparatively small model instances.

Parallel CPU implementations

There exist several parallel implementations that run on CPUs. A frequent challenge of these works is an inadequate method, for example, the First Reaction Method [149]. The superfluous work these methods do (compared to more advanced dependency-aware methods) is somewhat more accessible to parallel speedup. More advanced methods for parallel CPU execution try to partition the model. Nonetheless, the challenge of frequent synchronization remains. A partitioning method for the Next Reaction Method [150] approaches this with a master-slave architecture that synchronizes every step. Others have introduced a new approximate algorithm [151] that reduces the number of synchronizations. There has also been some success in applying FPGAs to this problem [152]. FPGAs (or field-programmable gate arrays) are integrated circuits whose structure can be reprogrammed, yielding capabilities similar to custom embedded hardware.

GPU-specific challenges

When using GPUs, particular technical challenges have to be considered.

GPUs achieve the highest performance when executing programs with largely homogeneous control flow and memory access patterns, which runs counter to the sparsity and heterogeneity of the computations involved when executing various parametrizations and replications of stochastic simulations in parallel. Thus, GPU-based SSA implementations require careful consideration of the GPU's execution model and hardware properties. A frequently employed means of optimizing execution speed is the efficient usage of the GPU's memory hierarchy. For instance, [153] analyzes different data structures to store model parametrizations

in fast read-only regions of GPU memory. Consideration is also given to dense and compressed storage of data [154]. The issue of thread divergence can be addressed by sorting the computations involved in different simulation replications so that adjacent threads execute the same control flow [155].

The Direct Method is used in several GPU implementations [156,153,148,157], as its relatively simple control flow makes it reasonably straightforward to map to GPU execution. A similar, easier target is the First Reaction Method [158]. Another GPU-Simulator [159] uses the more advanced Logarithmic Direct Method.

Dependency graph-based methods perform better for larger models but are more difficult to implement and less prevalent in the literature. In [160], the Next Reaction Method and the Logarithmic Direct Method are implemented in addition to the Direct Method. There is also work on implementing the Optimized Direct Method [154]. Here, to enable additional fine-grained parallelization, propensity values and their respective sums are updated in parallel using a prefix sum algorithm.

Compartmental models

When modeling large systems, the assumption of spatial homogeneity at the core of the SSA abstraction is no longer appropriate. Rather conveniently, this also increases the amount of available parallelism. One intermediate approach between purely non-spatial SSA and fully spatial multi-particle simulation [69] is the Next Subvolume Method (NSM) [161]. Here, the system is divided into subvolumes, each of which contains its own population. Diffusion reactions occur among the species in adjacent volumes, corresponding to physical movement and balancing among subvolumes.

Implementations of NSM

An implementation of the NSM in the Aurora simulation environment [162] demonstrated the difficulties of achieving speedup in a distributed-memory setting, where an efficient parallelization across computationally fine-grained events is particularly challenging. The Abstract Next Subvolume Method parallelizes the NSM by mapping it onto the parallel discrete event simulation paradigm [163]. Different implementations of the Abstract Next Subvolume Method using Breathing Time Warp have been shown to outperform both Time Warp and Breathing Time Buckets [164]. Another approach to parallelizing the NSM is to design dedicated parallel algorithms tailored to the NSM's known characteristics. One example is to estimate the communication among simulation entities based on model-specific knowledge and to minimize the overhead by performing rollbacks only selectively. This approach has been applied to the All Events Method, in which transitions are more costly than in the NSM, making speedup more achievable [165], and later to the NSM [166].

Next Subvolume method and Window Racer

As described above, synchronization algorithms for parallel simulation struggle to attain speedup if the simulation entities are tightly coupled and their interactions are difficult to predict. Window Racer [15] is a parallel synchronization algorithm for shared-memory architectures specifically targeted toward attaining speedup in these challenging cases. The key idea is to speculatively process sequences of dependent events even across partition boundaries through fine-grained locking and low-overhead rollbacks while negotiating a global synchronization window that rules out transitive rollbacks (See Figure 3).

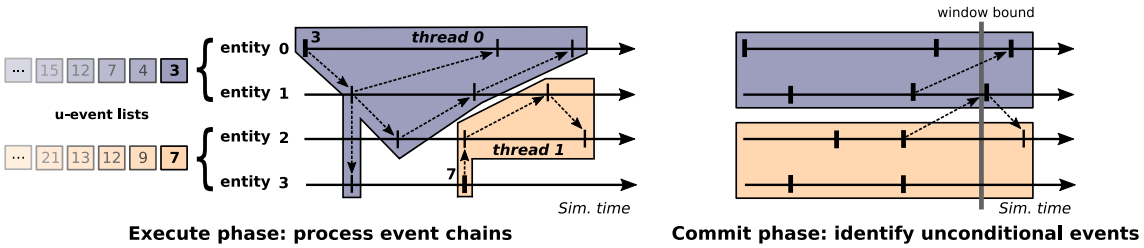


Figure 3: Illustration of the Window Racer Algorithm from [147]. Each thread maintains a *u-event list* holding unconditional events for a subset of entities. In the *execute phase*, threads process event chains generated by local entities' unconditional events regardless of the thread assignment of newly generated events, up to a dynamically computed window bound. In the *commit phase*, each thread identifies local events that have become unconditional, which are inserted in the thread's u-event list for the next iteration.

We have implemented the Next Subvolume Method (NSM) within the Window Racer algorithm. In a first experiment, a reaction network with a low degree of coupling among subvolumes results in only one in five hundred events crossing the subvolume boundaries. We observe that Window Racer outperforms the sequential simulators for larger numbers of subvolumes. Beyond 2000 subvolumes, the speedup is attained over all sequential simulators with 16 and 32 threads. Further experiments were conducted to analyze the behavior of systems with higher diffusion, i.e., a greater coupling. Overall, we observed that the NSM implementation Window Racer could substantially outperform sequential simulators for spatial biochemical reaction networks given sufficiently large network sizes. Naturally, the benefits seen in practical studies will depend on the need for such extensive scenarios and on the degree of coupling in the modeled system.

Ensembles on GPU

In [D], we explored the challenge of executing simulation ensembles of stochastic reaction networks on a GPU. We have used a method similar to the optimized direct method (ODM) [140] described above, sorting reactions by their propensity, making the linear weighted choice algorithm faster. We have introduced two concepts that set this implementation apart from the previous work: *Computation Sorting* and *Code generation.*

Code generation

Traditionally, SSA simulators adhere to a strict separation between the simulator and the model. A generic simulator implementation handles functionalities such as the updating of propensities during a reaction, with dynamic concrete control flow and memory accesses depending on the model. Code generation offers the opportunity for compiler optimizations based on full knowledge of the simulator *and* the model. We will discuss the implications of model-specific code generation in a CPU setting in Section 3.5. In the GPU context, static optimizations of the memory access patterns may increase the opportunities for memory access coalescing. The separately generated code for each reaction comes at the potential cost of an increased divergence. On the GPU, threads are grouped into sets of hardware-dependent sizes of 32 or 64 threads, referred to as warps or wavefronts. Within a warp, threads operate in lockstep. Hence, if the control flow of the threads within a warp diverges, all branches are taken in sequence while discarding unneeded results. In addition, if adjacent threads access adjacent locations

in memory, the accesses are *coalesced* into a single memory transaction served as one.

Computation sorting

In a given simulation step, the different replications may execute different reactions, which causes GPU threads within a warp to diverge in both the execution and the update steps. Computation sorting aims to minimize divergence by sorting the replications by the reaction they choose in each simulation step. The basic simulation algorithm and its implementation remain unchanged. However, simulation replications are allocated to the threads in a sorted fashion according to the selected reactions. This increases the likelihood that identical reactions occur in adjacent threads, reducing divergence. The sorting introduces additional computational and memory access overhead. However, it should decrease thread divergence because most consecutive threads execute the same reactions as long as the number of possible reactions in the system is significantly smaller than the number of threads times the number of replications.

Performance of the GPU implementation

We consider four models to test the suitability of the different simulators and to obtain a comparison to the sequential implementation. Generally, we observed that full utilization of a modern GPU's computational resources is achieved only at large replication counts between 100 and 1000, depending on model size. In the best case, a GPU provided the equivalent of around 40 CPU cores executing an optimized CPU implementation. We find that for some problems, the computation sorting reduces the execution time of the simulation step substantially. However, this speedup does not outweigh the overhead introduced by the sorting. Future work could explore a localized sorting that would reduce control flow divergence while inducing lower overhead. Due to issues with their public availability, we could not directly compare to existing GPU implementations. We did, however, implement a basic GPU implementation of the Direct Method, which performed well for very small model instances but, as expected, did not scale with increasing instance size. An attractive avenue for future work is the exploration of improved memory layouts using thread-local storage, which exhibited promising results in our initial experiments.

In the next section, we will discuss our own *sequiential* variant of SSA, which we used for our CPU implementations.

3.4 Sorting Tree Direct Method

Custom SSA implementation

Based on hands-on experience, we have built our own variant of a direct method algorithm implementation based on a cumulative sum tracking tree. The groundwork for this method was laid in multiple smaller works of ours [3,6,8]. Later, we used closely related implementations both for agent-based simulation [F] and dynamic structure SSA [E]. In this section, we will give an overview of the algorithm and implementation. Some performance metrics will be given in Chapter 5.

Goals of our SSA implementation

The algorithm was also designed for larger systems. Therefore, a dependency graph is vital. Furthermore, we observed how, in the case studies where our implementation was used, a small set of reactions dominated the runtime. We wanted to optimize for these code paths. Finally, the goal was for logarithmic (in the number of reactions) step cost. While in theory the Next Reaction Method meets these criteria, we found that in practice, the underlying data structures and

procedures had a high constant complexity factor. Similar observations led to the development of the optimized direct method (ODM) [140]. We have, therefore, called it *tree-odm*. It also has some relation to the logarithmic direct method [142]. As many different SSA variants exist, we can not be sure that all ideas are unique. However, we believe their combination is novel.

Tree structure

The implementation is based on storing the propensities in a binary tree with cumulative sum tracking [3]. We were, in part, inspired by prefix sum implementations on the GPU [167]. Each node in the tree corresponds to one potential reaction and stores two values: The propensity of that reaction and the sum of all propensities of its child nodes. This allows for updates of the total propensity sum with logarithmic complexity for changing a single propensity. Furthermore, reaction selection (a weighted random choice based on the reaction propensities) can also be completed in logarithmic time complexity. Similar approaches are used in other algorithms like the Logarithmic Direct Method [168] or the simulator implementation for $S\pi@$ [169].

Numerical stability

An additional advantage of the tree-based approach is its improved numerical stability due to the fewer operations which are needed to update the propensities. Another way to minimize total propensity calculation time is by keeping track of the total propensity sum and only adding and subtracting the changes to reaction propensities after reaction execution, as done by the Optimized Direct Method [140]. However, numerical errors from floating-point addition and subtraction accumulate over time and need to be dealt with. In our tree-based approach, numerical rounding errors only originate from the single summation, similar (arguably even better) to a single linear summation approach. The numerical error does not accumulate over time. It is independent of the number of steps and depends only on the number of reactions. Another possible approach to reduce numerical problems could be Kahan summation [170].

Propensity update

After a reaction has fired, multiple propensities in the tree must be updated. In principle, each update in the tree could be done individually based on the dependency graph via updating the cumulative sum until we reach the root. Instead, we roll these changes out in two phases. First, the node values are changed based on the dependency graph. Afterwards, the cumulative sums are updated as needed. If one were to update all sums individually, some nodes near the root might perform repeated summation updates. What total summation operations need to be done for each reaction is calculated ahead of time in a static optimization phase. This computation can be accelerated via the use of bitsets.

Reaction selection

Reaction selection uses a weighted random choice based on the reaction propensities stored in the tree. The performance of reaction selection is improved by sorting the more likely reactions towards the root of the tree. I have explored using higher-order trees in [3], and this can lead to a small constant factor improvement when updating propensities but increases the cost of selecting a reaction. In that earlier implementation, some numerical error also accumulated from the summation across child nodes.

In the next section, we will further optimize by automatically adapting our simulator implementation to a specific model.

3.5 Simulator specialization

Compilation vs. interpretation

An external domain-specific language that is interpreted at runtime has a similar tradeoff to the interpretation of a scripting language. An overhead is introduced from this interpretation. This is the reason why native code is usually compiled: to increase performance, for example, through compiler optimizations and static (type) analysis.

Limitations of compilation

If compilation is more efficient, why do interpreted languages exist? Compiling is not always more efficient and, in many cases, not even practically possible. Let us take a look at the concept of compilation in general and how these effects relate to simulation. We will focus on the reasons scripting languages are not compiled in the context of this work. The first reason is *efficiency and responsiveness*. Compilation can increase program speed. However, time is taken for compilation itself. This can be quite substantial and introduces a latency of execution. In cases where programs are already "fast enough" when interpreted or only run a few times, this tradeoff is not worth it Secondly, not all languages are *suitable for compilation*. Dynamic typing is a big challenge, as is dynamic interpretation or adding source code. Also, there might be information available only at runtime that only then allows for optimal compilation [171]. Finally, there might be technological concerns, like web security. In Section 5.1, we will talk about WebAssembly as an intermediate representation for web deployment that addresses both security and performance challenges.

Dynamic compilation

How can we achieve compilation for Domain-specific language? The go-to tools in this domain are language tool benches [172]. For the Java Virtual Machine world, this is relatively easy. Java as a language is always interpreted and compiled at runtime. Having such a runtime simplifies the creation of dynamic code elements. This has been applied to part of the Stochastic Simulation propensity calculation [173]. Interestingly, this intermediate representation (Byte-code in Java world) has a lot of parallels to the independent description of GPU kernels in the form of PTX kernels [174] as used by CUDA. For native compilation, the most popular approach is using LLVM [175]. LLVM is a compiler toolchain. Many languages generate a compatible intermediate representation, which LLVM can utilize to optimize and then generate appropriate assembly. The concept of intermediate representations to map algorithms and implementations to various hardware has recently gained broader traction through its use in the optimized execution of AI workloads [176]. Generating optimizations for Stochastic Simulation in cellular biology via LLVM has also been successfully implemented in libroadrunner [177]. We choose to use Code generation to create a *Simulator Generator* tool [A], as discussed in the next paragraphs.

Partial evaluation

The idea of partial evaluation dates back at least to Futamura's work in the early 1970s [178]. Futamura describes different levels of program specialization, starting with some generic computation and gradually becoming more specialized. The key idea is to provide a computation α to automatically increase the level of specialization by partially evaluating a computation for some of its inputs.

Concretely, a computation π with $m + n$ inputs

$$\pi(i_1, ..., i_m, i_{m+1}, ..., i_{m+n})$$

can be specialized by supplying values for the first m inputs to α, resulting in a new computation π' that still takes the remaining n inputs.

$$\begin{aligned}\pi(i_1, ..., i_m, i_{m+1}, ..., i_{m+n}) &= \alpha(\pi, i_1, ..., i_m)(i_{m+1}, ..., i_{m+n}) \\ &= \pi'(i_{m+1}, ..., i_{m+n}).\end{aligned}$$

We can apply the same idea to the notions of generic and specialized simulators. A generic simulator sim is a computation that takes a model m, model parameters $p_1, ..., p_n$, and a random seed r as input and computes some sort of simulation output. By considering the model m as a static input and the remaining parameters as dynamic, we can obtain a specialized simulator via α. The computation that generates a specialized simulator for a given model (and a static generic simulator) is obtained with one additional level of specialization.

$$\begin{aligned} & sim(m, p_1, ..., p_n, r) && \text{Generic Simulator} \\ = \; & \alpha(sim, m)(p_1, ..., p_n, r) && \text{Specialized Simulator} \\ = \; & \underline{\alpha(\alpha, sim)}(m)(p_1, ..., p_n, r) && \text{Simulator Generator}\end{aligned}$$

Similar to how a compiler works, a simulator generator is able to introduce model-specific optimizations into the generated simulator. In particular, it can exploit the properties of the model to speed up the simulation. Typically, these properties can be determined by syntactic or semantic analysis of the model. For example, a generic algorithm must be able to run models with arbitrarily large states and, therefore, use a flexible data structure. A specialized algorithm, on the other hand, can hardcode the state representation and avoid indirections when accessing it.

Related approaches

In practice, partial evaluation has been applied to speed up computations, often by simplifying them. For example, Schultz et al. [179] present an approach that employs partial evaluation to remove object orientation from Java code to increase computational performance. In a similar fashion, Zeng [180] applies partial evaluation to remove concurrency or recursion from programs written in domain-specific languages (DSLs). Other works focus on developing methods for effectively integrating partial evaluation into programs. Rompf and Odersky [181] present a library that exploits Scala's expressive type system to distinguish expressions evaluated during program generation and during program execution. Leißa et al. [182] presents a DSL that allows partial evaluation targeting CPUs as well as GPUs and apply it to computation-intensive problems like image processing or genome sequencing.

Performance results

We have implemented multiple versions of this simulator generator targeting different programming languages. One key result is shown in Figure 4, where we have taken 4 different models of increasing size and explored their performance in detail.

We can see how the different optimizations affect different phases differently for different models. The models are sorted by increasing size from left to right. There are different ways to conduct the profiling, leading to two similar, but not identical results for each experiment. Shown is the data for two different optimizations. We have explored both the reaction sorting of the ODM (Section 3.2) and

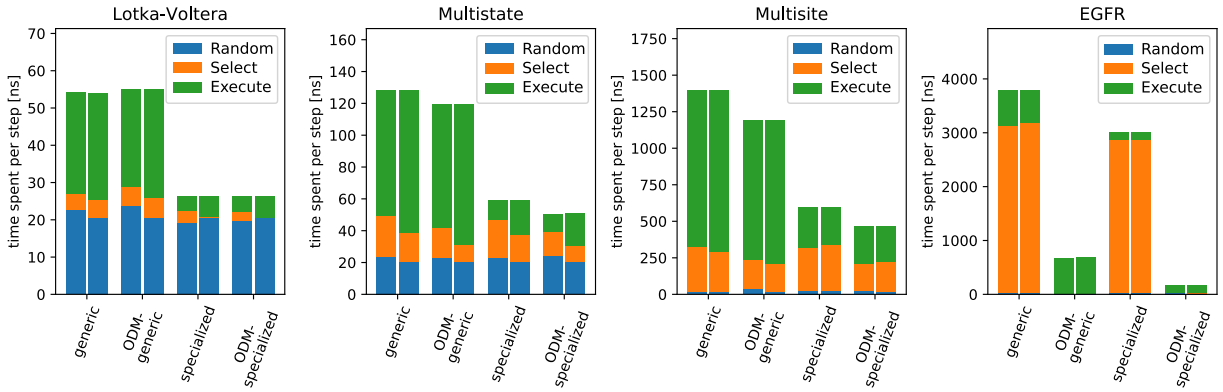


Figure 4: Profiling results from [A] comparing a generic simulation code to that of a specialized simulator created through partial evaluation. Shown is the time spent on the three phases of the direct method style SSA (random number generation, reaction selection, and reaction execution, see Section 3.2).

the Simulator specialization (i.e. the simulator generated through our simulation generator).

Random number generation

Smaller models are significantly influenced by random number generation, whereas random number generation has little effect on larger models. This is because random number generation takes on the order of 20 ns, independent of the size of the model. We notice that for the case of the extremely small model (the Lotka-Volterra model), the time for the random number generation slightly decreases in the specialized simulator. This could potentially be due to less overall simulation effort being spent elsewhere, resulting in more of the random number generator staying in the cache.

Reaction execution

Reaction execution is affected the most by specialization. When executing a reaction, the generic simulator needs to do a lot of work to look up what to do, for example what species to alter and what rate constant a reaction has. Furthermore, the dependent reactions also need to be updated. The specialized simulator has all this hard-coded in the assembly. The observed speedup in the execution phase is around 6x for the models tested.

Reaction selection

Reaction selection only plays a minor role for smaller models. On the other hand, for the larger models we tested, it can be the dominating factor in runtime performance. The reaction sorting as done by the ODM can provide speedup of up to two orders of magnitude for the selection phase of the largest model tested.

Degrees of specialization

We also varied the amount of partial evaluation performed by introducing lesser levels of partial evaluation. The idea is to still have some benefits of partial evaluation with less up-front cost, making partial evaluation also accessible for larger models, where the huge code size of the generated binary is a limiting factor. Our results here mainly indicate that it is usually not worth it. The lower levels of partial evaluation provide only little practical use. We did, however, gain insight into the partial evaluation of the dependency graph, which is what explains the performance of the specialized simulator.

When considering large models, generating a single file of source code is impractical due to large compilation time and memory needs. We, therefore, have explored strategies to mitigate this. We found separation into different compilation

units to be helpful at improving compilation time when compiling on a parallel machine. Link-time optimization provides an intermediate solution that somewhat increases compilation time but also improves execution time. Nonetheless, the performance of these faster to compile variants is not as great as for the single-file variant because reaction execution now stretches (potentially) over many compilation units.

3.6 Summary

In this chapter, we have discussed reactions and their simulation in the context of Gillespie's Stochastic Simulation Algorithm for population dynamics. We have seen how this abstraction can be simulated efficiently on parallel hardware (CPU and GPU). Finally, we discussed different approaches for efficient sequential execution, like the tree-ODM method and partial evaluation.

The next chapter will now take the methods presented here and expand on them, adding additional functionality or expressiveness. These will enable new domains but also bring new challenges for efficient execution.

4 Dynamic Structure Models

In the previous Chapter 3, we discussed abstractions in general and Gillespie's Stochastic Simulation Algorithms in particular. So far, the number and structure of the reactions have been considered constant. In this chapter, we will add a dynamic structure to these static models. We will consider three types of dynamic structure modeling approaches and their efficient execution.

First, we will introduce the concept of dynamic hierarchy (i.e., nesting) rules, which, when applied to SSA, leads to the ML-Rules language and formalism (Section 4.1). Next, we will discuss our work on integrating dynamic nesting into a spatial simulation approach called ML-Force (Section 4.2). Finally, we will take the fundamental mathematical CTMC formalism and build from it a simulation tool called ML3 for agent-based simulation in Section 4.3 and discuss its efficient simulation.

This chapter includes work, results, and text (copied or adapted) from the following of the author's previous works: [4], [17], [B], [E], and [F].

Fixed structure

A fixed size and structure of the simulation model are common across simulation domains and have several advantages for efficient execution. A fixed structure can be laid out more efficiently in memory, and neighborhoods or other locality-dependent computations give constant results. For example, in factory simulation, if the factory layout is a fixed parameter, the paths between the machinery only need to be computed once and can be reused thereafter. Implementing fixed structure models is also less error-prone and easier to reason about.

A dynamic structure requires many more dynamic computations. Imagine, for example, a dynamic road network where the roads change throughout the simulation. Agents regularly need to update their route planning, and an optimal route will be different at different times in the network. Optimizations, like the code generation and partial evaluation outlined in Section 3.5 are both more involved and less effective in these scenarios.

Motivating compartments

In the realm of cellular biology, compartmentalization is essential for the modeling of many processes [183]. Compartments affect cellular processes by controlling both the reactions that can occur and the rate at which they do. In Figure 5 we see an illustration of such a process, where during the endocytosis of a lipoplex, an endosome is formed around it. Dynamic compartments are considered important to modeling signaling pathways, in which extracellular ligands trigger a cascade of biochemical reactions that span various cellular compartments, such as membrane, cytosol, and nucleus [184]. A prominent example is the canonical Wnt/β-catenin signaling pathway, which is essential for cellular functions such as proliferation and differentiation and is involved in several diseases, including cancer [185]. In the last 20 years, more than 20 quantitative models have been built to analyze the Wnt/β-catenin signaling pathway [186]. Many of these models express the causal influence of compartments on processes, e.g., constrain processes to specific compartments or study proteins shuttling between compartments [187,188]. Also, the volume of the compartment may influence the reactants' density and, consequently, the kinetics within the different compartments [189,190]. Compartmental constraints on reactions are supported

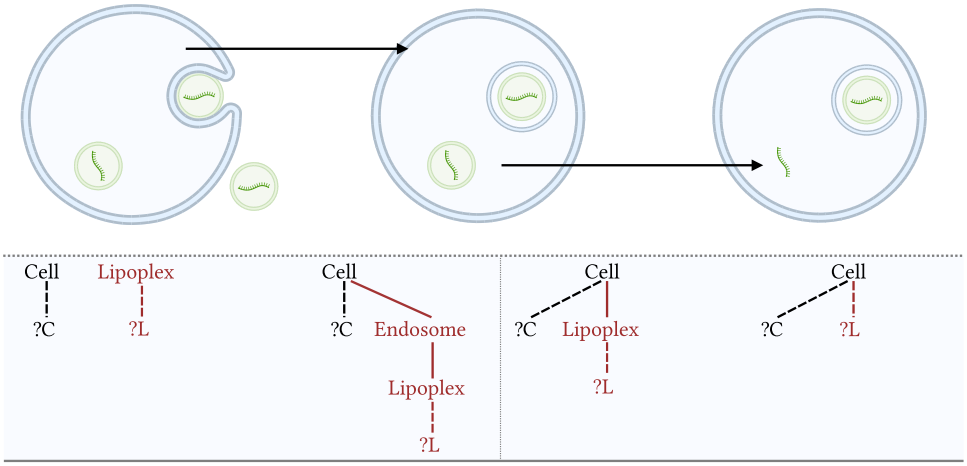


Figure 5: Illustration from [E] created with BioRender showing *creation and removal of compartments*. During the endocytosis of the lipoplex, an endosome is formed around it. Inside the cell, the lipoplex can unpack its content (?L) into the cell. In the top panel, we find a graphical representation. The bottom panel contains the logical nesting hierarchy. Denoted in red is the moving part of the system.

by most simulation tools and standardized modeling exchange formats such as SBML [190].

4.1 Dynamic rule-based models

Dynamic SSA

Before we can consider dynamic hierarchies, we need to consider the concept of *rule-based modeling* [191]. In the previous sections, we have seen reactions that only specify which species' population to increase or decrease. Similar to objects in object-oriented programming, species may also have attributes. Changes in the attribute values can be expressed as mathematical functions (depending on the expressiveness of the simulation tool used). The fundamental idea of rule-based modeling is to use rewriting rules to specify transition classes. If the number of potential species variations (i.e., attribute value combinations) is small enough, a complete set of reactions may be generated from the rules as a static *reaction network*. For these reaction networks, the simulation approaches discussed in Section 3.2 may be used. This capability is the name-giving feature of the Biological-Network-Generator BioNetGen [191].

Network-free simulation

When enumerating all possible values of attributes in advance is not possible, network-free methods can be used [192]. Here, after every reaction execution, changes to the system are dynamically monitored. If novel attribute values are created, new possible reactions are initiated. This requires regular restructuring of the data structures. The dependency graphs, which we saw in the previous chapter to be a powerful element of efficient SSA, are no longer static. These dynamic checks incur significant runtime costs, making them challenging to execute efficiently [71].

Dynamic hierarchies & ML-Rules

Static hierarchies are frequently implemented for existing SSA tools, both rule- and reaction-based [81]. However, dynamic nesting (where any object may dynamically change its content) is a lot more challenging. One approach to dy-

namic hierarchies is ML-Rules [193]. ML-Rules allows the description of dynamic nesting processes using rules similar to the reaction-generating rules used for attribute values. Key to a concise description of dynamic structure processes is hierarchical rewriting rules, specifically introducing the notion of a rest variable into biochemical rules. A rest variable is a syntactic expression describing what happens to all entities that are not explicitly named in the rule [194]. The details of the ML-Rules specification are discussed in its formal semantics [195]. In Figure 5, the compartments' contents are bonded to variables starting with a question mark, which are the rest variables.

Efficient multi-level simulation

The efficient execution of dynamic changes in the system's structure has been a challenge in the past. Most simulation tools assume static compartmental structures and their simulators have been optimized accordingly [196,197,81]. To integrate compartmental dynamics into these simulators without significant loss of efficiency has been identified as a daunting challenge [198].

To solve the problem of dynamic structural changes, one option is to write a fully dynamic simulator, which traverses the tree structure of the model after each reaction execution to instantiate new reactions and re-calculate propensities on demand, and also checks whether the dependency graph requires any updates. This is how a previous implementation of ML-Rules was realized [199]. However, compared to methods optimized for static networks [71], this has a significant overhead.

New hybrid approach

To improve efficiency, we implemented a new hybrid simulator approach. During an additional analysis step, every reaction is analyzed and marked whether its execution would result in a structural change. In the models we encountered, most reactions do not induce a structural change. These reactions are called regular reactions. The rules altering the dynamic structure are called complex. A similar distinction between regular and complex transitions has been made when simulating parts of the system using deterministic numerical integration methods [200] or by τ-leaping [201].

The primary mode of our simulator is running a static SSA (as described in Section 3.4) that executes only regular (non-dynamic) reactions. The model representation has been flattened, enabling static index-based access to the model's state. Every species' amount is stored in an array. Hierarchical relations are only preserved to the point where they are needed to reverse the system representation to a tree form. However, they are not needed during the simulation as long as no transformation to a tree form is required. When we encounter a structure-changing reaction, the model is transformed back into the tree representation reflecting the compartmental structures. The reaction is applied to this structural representation of the model. The resulting structured state is then transformed into a flat representation, and processing is continued. Every dynamic structure change requires an entire rebuild of the simulator, including re-enumerating all potential reactions and a do-over of the static optimization phase. Especially for larger systems, this can be relatively costly. However, as long as dynamic structural changes are rare, the costs amortize.

We foresee multiple approaches to address the high transformation cost [E]. If changes to the dynamic structure are localized, moving forward, only a subset of

the simulator could be transformed. It would also be possible to integrate the dynamic structure more closely into the simulator at the cost of some performance for the regular transitions.

Challenges of partial evaluation

We saw in Section 3.5 that SSA simulations can immensely benefit from partial evaluation. There, we developed a specialized simulator generator for reaction-based SSA, which is a sub-class of ML-Rules. Adopting this approach for dynamic compartments appears impractical. Its performance gain relied to a large degree on optimizing the updates of the dependency network. With dynamic compartments, these reaction networks change during execution. Thus, optimization (i.e., recompilation) would need to be repeated after each structural change. This recompilation would induce a significant overhead. Especially for large models (which are typical for ML-Rules models), the additional compilation steps would increase runtime significantly, independently of model execution duration. Therefore, we developed an approach that combines dynamic interpretation and partial evaluation and does not require repeated compilation on every change. Our approach focuses on the rate evaluation (and thus propensity updates). This is costly for most dynamic compartmental models, as rate expressions involving dynamic compartments are typically complex. The complex expressions result in large abstract syntax trees (ASTs) that need to be parsed during execution. When using an interpreted or reflection-capable language like Java, code generation for these expressions can be introduced relatively simply [173]. A different approach is required for compiled languages.

Performance templates

Our approach, which we developed for ML-Rules, generates performance templates during the simulation and stores them for later reuse. Every time the simulator encounters a new AST, it checks whether generated code in the form of a performance template is available to replace this tree. Each performance template represents a partially evaluated AST and can be parameterized with numerical values (like model-specific constants) and indices (as used to identify involved species). If such an optimized previous version is found, it is used instead of the AST. If not, the simulator dynamically interprets the expression (by evaluating the AST). It generates an optimized code for future use and stores the corresponding performance template. With every new model developed or recompilation of the model, the simulator can reuse previously created templates.

4.2 Spatial dynamic compartments

New method needed

Compartmentalization is inherently spatial. An object's outer compartment constrains its location. As we saw in Section 2.2, constraining populations to a grid in order to approximate spatial processes can be cheaper than simulating individual particles (e.g., via the Reaction-Diffusion Master equation [202] in the Next Subvolume Method [161] we saw in Section 3.3). We are, however, interested in methods that can describe the abstract processes discussed above. Specifically, we would like to describe processes of dynamic hierarchical nesting. One such method is ML-Space [93,203]. ML-Space considers the case of the overdamped Langevin equation, where all momentum is immediately dissipated [204]. This makes it very challenging to include external forces in the model. Other challenges include the handling of crowding, where a rejection scheme of movement was used. However, its underlying mathematics was not explored in detail.

ML-Force abstraction

At this point, we can look back at the Takahashi and Schönberg Classification schemes from Section 2.2. The literature lacks a method that combines dynamic compartments with spatial forces. We have identified this gap in capabilities and developed ML-Force[1] [B]. Following a bottom-up approach, ML-Force builds on the ideas of particle simulation in Molecular Dynamics. All transitions, including creation, merging, splitting, or entering of compartments, occur in a smooth, continuous manner. ML-Force is a particle-based approach. The particles are hollow spheres. As particles can contain other particles, they can be used to model cellular compartments. For particles to interact, they need to be in close proximity. In ML-Force, we base our model of propagation on the Langevin equation [205]. ML-Force combines the Takahashi classification's *a*,*b*, and *c* capabilities at *level 4* of the Schönberg description.

Potential-based methods

Crowding has a significant effect on the physicochemical kinetics of various intracellular processes [208]. Space exclusions need to be considered to capture the effects of molecular crowding. In addition, interactions based on potentials are an effective means to study the formation of clusters [91]. Additionally, potentials allow us to capture directed movements, such as the transport of vesicles [209]. Similar to ReaDDy [90], SRSim [91], or SpringSaLaD [89], ML-Force uses potentials for particle-particle interaction.

Compartmental methods

Intra-cellular space is further structured by compartments and vesicles, most of which are subject to frequent changes in terms of their numbers, contents, and interconnectivity. ML-Force uses the same force-based approach for the interaction of particles and global force functions to model the compartmental dynamics. Similar to ML-Space, ML-Force supports both compartmental and particle dynamics. However, unlike ML-Space, it uses potentials.

ML-Force rule-based

In ML-Force, the dynamics of particles and their interactions can be defined by rules. Here, ML-Force supports arbitrary unconstrained attributes. This includes rules for bimolecular reactions like dynamic nesting. Since every particle is unique due to its unique location in space, a population-based approach is not possible at the particle level. This simplifies the realization of arbitrary attributes compared to the approaches described above, as each individual can track its own attribute values. However, the rule's expressive nature allows for population-based, non-spatial reactions to take place within each particle.

Realization of DSL

The simulator has been implemented using `C++`. The user interface is built around a rudimentary embedded domain-specific language. The developed embedded DSL bears some similarities with the ℓ-language [210], which also supports an imperative rather than a declarative approach to modeling.

The modeling layer makes heavy use of anonymous functions in order to encapsulate model-specific behavior into single function calls for the simulator. This is facilitated by various templates that have been defined in the DSL. For example, after each reaction, the simulator invokes the *post reaction function*. In the model-

[1] The name *ML-Force* is a word play, based on it sounding like ML4s, because it is the 4-th method developed in this family following ML-Rules [193], ML-Space [93] and ML3 [206]. This naming was originally motivated by eluding to *multi-level* models or *modeling-languages*. In hindsight, it is not ideal. Originally, it was frequently assumed to be related to the ML family of programming languages [207]. Nowadays, a relation to machine learning and AI is assumed.

ing layer, this function is specified by the user and denotes the results of applying a rule, such as particles changing state, being degraded, or being created.

This simplifies the implementation of the simulator and gives a unified and lean interface for implementing further features. The simulator also provides a set of units to the modeler, allowing for consistent parameterization. In general, the DSL is very expressive, as all rates and reactions are described as functions. These can be of arbitrary complexity. However, for many of the relatively simple uses within the ML-Force models, they are somewhat complex. The syntax is verbose for simple cases. For example, even a constant rate requires the specification of a (syntactically heavy) anonymous rate function. The macro approach we used in `C++` makes an embedded DSL possible, but it is very limited in its ergonomics compared to many more modern languages with advanced macro features like matching or types. Nonetheless, the functional approach is very capable, and even complex behavior could be added without additional language features needed. It served as an important starting point for our work on other embedded domain-specific languages, like the one we present in the next section.

Performance and GPU

One of the challenges of this method is the large computational effort. In part, this could be mitigated by offloading the challenging spatial intersection calculations to the GPU [2]. Here, we identified a novel collision detection problem in a hollow sphere intersection. We found that considering this specific hollow sphere problem outperforms state-of-the-art approaches for solid spheres in dense scenarios.

4.3 Continuous-time Markov chain for Agent Simulation

In this section, we apply the simulation concepts of the previous section to a different domain. Having looked at cell-biological applications so far, we will now use the methods for agent-based modeling [211]. In particular, we will explore this under the constraints of ML3, a modeling language for linked lives [206]. First, I will discuss the distinctions between this approach and the previously discussed formalisms. Next, we will identify the challenges in efficient computation and discuss some of our solutions.

Differences from cell biology

For agent-based simulation, we are concerned with individuals instead of populations. The motivation for developing the *Modeling Language for Linked Lives* (ML3) originated in social science applications. In ML3, individuals do not exist in isolation, but their *lives* are *linked* through social networks or other connections. This observation has led to the need to succinctly describe the diverse decision processes of such linked lives in continuous time [212]. The context for individuals to make decisions is formed by their social network. Further requirements are behaviors conditional on agent attributes, such as age-dependent behavior and stochastic waiting times [213]. Alternative decisions can be modeled as concurrent processes that compete by stochastic race [212].

In ML3, agents represent any entity in the model, including individual persons, but also higher-level actors such as households, towns, governments, or other policy-makers. Each agent has a *type*, and the type determines its attributes and behavior. The type corresponds to the species in ML-Rules.

The relations between agents in ML3 are represented with *links*. Links are declared between agents of a specific type. They might refer to social ties, linking

an agent to friends or an agent being part of a household. Links also allow the introduction of discrete locations, for example, by linking agents to the town where they are located. Such connections are always on an individual basis, not at the population level. From an ML3 point of view, the nesting structure of ML-Rules could be viewed as a directional tree structure of the links. In fact, ML-Rules is a subset of ML3. ML-Rules models can be expressed in ML3, however, with some rather inconvenient syntax because the chemical reactions, whose propensities are calculated based on the global state of the system, are quite involved to express in ML3.

The rules in ML3 are inspired by guarded commands. They consist of three parts: *guard*, *rate*, and *update*. This triple has been found to form a natural description for continuous-time population-based models [214]. Each rule in ML3 is assigned to a specific type of agent and specifies under which condition, at which rate, and which state changes occur. The *guard* condition is evaluated as a boolean variable. This is mostly a syntactic feature that allows for more convenient notation of conditional changes. The *effect* specifies the state change which occurs when the rule is executed. An agent's transition can affect not only its own state but also the state of other agents as well as links of its social network. Finally, the *rate*, an expression that evaluates to a positive real number, specifies the propensity of the transition.

Non-local effect

The effect of a transition in ML3 is very complex. Basic SSAs only allow for the change in populations in their effect. Rule-based modeling tools might allow for new attribute values (and thereby populations) to be created. In ML3, the effect is non-local and potentially affects many linked or globally accessible agents in the system. This is a particular challenge when working in parallel or distributed settings. S3A [215] (a precursor to the WindowRacer Algorithm discussed in Section 3.3) was developed specifically to work in these zero-lookahead dynamically connected agent scenarios. Zero-lookahead here means that there is no minimum time (i.e., a lookahead) between one transition and its effect on another agent. In parallel discrete event simulation [106], a non-zero lookahead is advantageous or even required for some methods. The global effect results in two challenges when building an ML3 implementation. First of all, we need a very expressive domain-specific language. Secondly, to use efficient algorithms for larger systems, we need a dynamic dependency tracking [216]. We will discuss approaches to these two related challenges in the remainder of this chapter.

Embedded language

The first ML3 implementation [213] used an imperative external domain-specific language (DSL). However, relatively quickly, the limits of creating and maintaining a custom language for the agent's complex behavior were reached. Therefore, we have explored the idea of using an embedded language where the host language provides a large ecosystem of functionality. On the other hand, it is easier to keep track of the dynamic dependency graph when interpreting an external DSL. This is because all accesses are mitigated through the simulation runtime and can be tracked therein. We do not have such facilities when using a host general-purpose language for the embedded DSL. We have approached this challenge of using a host language with all its features and still have dependency tracking in two different ways. First, we have a top-down approach, using aspect-oriented

programming to integrate within the Java virtual machine. Second, we take a bottom-up approach, using code generation for custom tracking interfaces within a native Rust code setup.

Aspect-oriented programming

Aspect-oriented programming is a programming technique that is used to maintain a separation of concern in cases of cross-cutting code [217]. The idea is to use a specific syntax to access so-called *join points*, where custom code is injected. We realized this on top of the existing MASON framework for Agent-based simulation [218]. This allows the modeler to use the host programming language (e.g., Java) the way they are accustomed to it. All accesses and tracking happen at runtime through aspect-oriented programming. We found that aspect-oriented programming introduces some additional overhead, but for most systems, where updates are sparse, this overhead was far outweighed by the overall reduction in computational effort [17].

Interface generation

When peak performance is needed, native programming languages are at an advantage. Our goal was to build a high-performance implementation of ML3. We built it using the rust programming language [F]. It was our goal to keep the external language close to the host language's look and feel. To realize this, we employed compile-time code generation of interfaces that the simulator uses to keep track of the dynamic dependency graph. The interface consists of object-oriented getters and setters.

We have benchmarked this new native rust implementation alongside a time-stepped NetLogo [116] implementation, a Java implementation of ML3 [213], and a custom model-specific implementation [F]. For this benchmark, we used a model of infection spread and found that the new rust implementation performed very well. The optimized custom implementation was faster. However, that custom implementation was made specifically for that model, with a strong focus on efficiency and without concerns for the model's usability or extensibility. The NetLogo implementation performed well for large timesteps. However, this stepwise execution introduced a significant error (see Section 2.3). The ML3 Java implementation using an external domain-specific language was very inefficient (more than 20 times slower for small systems). In fact, in that implementation, the step cost scaled linearly with the number of agents in the system, whereas it scaled logarithmically in the new rust implementation (due to the tree structure described in Section 3.4), making the difference even bigger for larger systems.

4.4 Summary

In this chapter, we have introduced dynamic structures on top of reactions. The three realizations (ML-Rules, ML-Force, and ML3) differ in their domain and the realization of their domain-specific language. In [4], we've compared the approaches for ML3 and ML-Rules 3. Some differences stem from the differences in handling external and internal DSLs. For efficient execution of the external DSL ML-Rules, we needed several passes of analysis of the involved reactants and rule structure to minimize the work done during execution. Such analysis is relatively easy for an external language, where we have simple access to the entire syntax tree. However, the efficient evaluation of numerical expressions required additional work. This posed no problem for the internal DSL ML3. Here, it was more challenging to access and analyze dependencies to efficiently execute the model

during simulation. Finally, for ML-Force, the challenges of efficient execution lie entirely with the simulation engine and its spatial computations, where we used GPUs to improve the throughput on the inherently parallel spatial computations.

5 Evaluation and Replication

A key component to discussing the principles of models and their efficient simulation is their evaluation. In this chapter, we elaborate on the evaluation and replication of stochastic simulation tools.

Initially, we discuss the problem of reproducibility and comparability of performance results in general (Section 5.1). Next, two approaches to particular performance evaluation problems we developed are shown: A concept for benchmarking parallel discrete event simulators (Section 5.2) and an evaluation study for Stochastic Simulation Algorithms (Section 5.3).

This chapter includes work, results, and text (copied or adapted) from the following of the author's previous works: [E] and [5] (including an unpublished extended version)

Reproducibility crisis

The term *reprodicibility crisis* has become common knowledge through several papers [219] that found significant issues with the ability to reproduce scientific results across research domains. It is also discussed, sometimes controversially [220], in the simulation community. The underlying reasons are manifold, and their discussion is beyond the scope of this thesis. However, I would like to discuss in more depth some technical challenges and their methodological remedies in the context of software performance in simulation.

In general, there are two categories of the problem:

- A lack of **reproducibility** and **replicability** where the results from a paper can not be independently reproduced [221]. We will discuss this issue in Section 5.1.
- A lack of **comparability** and **representativity** where the chosen benchmark does not provide a suitable comparison to the state of the art [222,223]. Our work to address comparability in two specific subfields is described in Section 5.2 and Section 5.3.

5.1 Reproducibility

To address the question of reproducibility of published work in Computer Science, the Association for Computing Machinery (ACM) suggested a policy of reviewing and badging associated software artifacts. Following the ACM reproducibility initiative, reproducibility initiatives have been discussed and rolled out across ACMs groups [224,225], including reports written for papers in the ACM Special Interest Group on Simulation [10–12].

Terminology

First, we need to define the terminology involved. We will follow the updated ACM classification, which in turn follows the international vocabulary of metrology [226] and the US National Information Standards Organization recommended practice on reproducibility badging and definitions[2].

A scientific result has been *replicated* if a different team, using a different experimental setup, can obtain the same results or conclusions as the original team. The slightly weaker constraint lies in *reproduced* where a different team may use

[2]Interestingly, the harmonization with the US National Information Standards Organization has led to a swap in the meanings of *reproduced* and *replicated*. Until August 2020, those meanings were the other way around. The new nomenclature is considered Version 1.1. However, many older publications use the older version.

the same experimental setup (in our case, usually a software artifact) to get the same results. Next, we will discuss technical approaches to improve the ability to replicate existing results through simplified deployment.

When executing an experiment, it is not always clear whether the original results have been reproduced. Dalle [221] introduces different levels of reproducibility for simulation ranging from L1 of deterministic, identical computation to more relaxed L4 where similar scenarios and instrumentation are considered reproduced. The requirements and technical needs for reproducibility vary in different simulation approaches and have, therefore, led to different reporting guidelines [227,228]. One aspect is the provenance of the model and experiment [229]. The automatic capturing and analysis of model provenance information and its use in reproducibility is a topic of ongoing research [230,231].

Deployment of artifacts

One key challenge of creating replicable artifacts is streamlining their deployment. Standard practice is to provide a script that automatically installs and executes the experiments. However, these scripts are highly dependent on their software environment, most prominently the specific operating system used. The most promising remedy to this is containerization [232,233]. Containers are an encapsulation of an entire software environment used to deploy specific software.

A related question about deployments is how the involved experiments are specified. The concept of abstraction through domain-specific languages, as discussed in Chapter 3, has successfully applied here for both models and experiment specification [234].

Web deployment

As deployment through web apps is prevalent across our everyday lives, it is natural to explore this also for reproducibility in simulation. The idea to run simulations on the web is not new and has been around almost as long as the web itself [235].

WebAssembly

Web deployment has traditionally not been suitable for the long-term reproducibility of artifacts because the actual computation execution has been typically delegated to a server in the background or the cloud to enable an efficient simulation of models [236]. Such services tend not to be hosted for a sufficient duration. WebAssembly is a recently developed binary format that enables fast execution in web browsers [237]. WebAssembly is executed on a stack-based virtual machine similar to other native code. Additionally, it interoperates well with JavaScript. Without major changes to the underlying simulator code, WebAssembly allows reasonable simulation performance for an end user requiring only a web browser [238].

For the Rust simulator of ML-Rules [E] (Section 4.1), we reused the same code base as the existing simulator and only needed to add a small text editor front. A two-panel user interface is provided at `mlrules.pages.dev`, which includes a code editor with some syntax highlighting based on the Monaco editor from Visual Studio Code and another panel to display simulation results or any errors and instructions from the simulator.

ML-Rules 3 on web

A significant advantage of this WebAssembly approach is that there are next to no hosting costs. The simulation is executed on the end user's machine. Further-

more, we can provide direct links to specific models, e.g., `mlrules.pages.dev/pp/10/day` or benchmarks, e.g., `mlrules.pages.dev/benchmark/yeast/3/1000`.

Potential of WebAssembly

WebAssembly has a performance penalty of only about 10% to 50% compared to native execution. This is remarkable considering its technological constraints. With the advances of WebAssembly and the growth in its ecosystem, we advocate an increase in its use in simulation science, furthering reproducibility and accessibility. This could include, to a degree, performance studies, however will initially be focused on simulation tools.

The deployment is not the only issue when publishing simulation performance results. In the next two sections, we will make suggestions on appropriate benchmarking models for representative and comparable results.

5.2 Benchmark Model for Parallel Discrete Event Simulation

Motivation

One of the primary motivations for parallelizing discrete-event simulations (PDES) is the reduction of execution times [239,107]. The significance of novel methods and implementations developed with this motivation depends on their performance on well-defined benchmark problems relative to state-of-the-art implementations. Other computational fields follow this mode of evaluation. For instance, the various libraries that implement the Basic Linear Algebra Subprograms (BLAS) specification [240,241]. Using these standards, the implementations can be compared regarding their throughput for matrix operations [242,243].

State of the art

Some synthetic benchmark models such as PHOLD [244], EPHOLD [245], and LA-PDES [246] or proposals targeted towards specific simulation approaches such as DEVSTONE [247] exist. Further, there has been progress in characterizing real-world workloads [248–250].

Still, the various model variants and parametrizations used in simulator performance evaluations make comparisons across studies difficult. Further, commonly employed benchmark models permit comparisons of the simulation results across simulators only on a statistical level.

Goals

It is our goal to create a synthetic benchmark model for PDES that avoids the common pitfalls of parallel reproducible simulation [251]. With our model, we aim for two specific goals, *reproducibility* and *representativity* .

Goal 1: Reproducibility

We aim to enable researchers to obtain comparable results when repeating published experiments, even across programming languages, hardware platforms, and simulators. To this end, we propose a fully deterministic synthetic benchmark model related to PHOLD and La-pdes. “Deterministic, identical computations” forms the strictest level of reproducibility, which demands “the ability to re-execute the exact same simulation history as previously in terms of computation, including the inherently non-deterministic computations” [221]. We achieve this strict level of reproducibility by handling pseudo-randomness and simultaneous events as part of the model, which avoids deviating results among simulators. At termination, a single checksum attests to the correctness of the execution.

Goal 2: Representativity

Performance evaluations should capture the performance properties of a reasonably comprehensive set of models relevant to practitioners without putting ex-

cessive weight on particular properties. To achieve this, the proposed benchmark model is configurable for various essential performance-affecting characteristics.

Challenge of determinism

Our benchmark model is fully deterministic. This simplifies testing the correctness of the simulation execution as well as the execution of the same model logic and the same numerical load. This includes pseudo-random numbers, which are generated as part of the model and not the simulator. By not prescribing the concrete realization of pseudo-randomness the specifications of existing benchmark models such as PHOLD and La-pdes do not guarantee determinism.

Mathematically speaking, the entire simulation could be viewed as a hash function on the configuration. This hash function comprises numerous interconnected deterministic 64-bit xorshift random number generators [252] Each entity is in itself a random number generator. When receiving an event, the local state is altered by the event payload, such that future random numbers also change. In effect, the overall system is comprised of communicating deterministic random number generators that alter one another's state. All computation is based on integer math to achieve full determinism independent of the language, compiler, or hardware used to avoid some of the nondeterministic aspects of floating point computation [109]. Therefore, we omit a full description of the code here but have created annotated and encapsulated versions of the source code for this purpose. The following will give an overview of our approach, in particular, our handling of the methodological challenges.

Model parameters

To represent various workloads, we have 10 model parameters, as outlined in Table 1. The model is initialized in a steady state due to the symmetric pattern of initialization and message sending. The system has a fixed number of entities. We define entities as indivisible simulation objects, for each of which events must be handled in non-decreasing timestamp order. There are no assumptions on the assignment of entities to logical processes (LPs), operating system processes, or threads. They are placed in a ring topology to enable a notion of locality. Each of the individual values that make up the state is interpreted as the state of a 64-bit xorshift random number generator [252].

Communication patterns

The entities communicate using messages generated from their local RNG state that affect the recipients' RNG state. The entities are placed along a ring structure to represent locality. Communication can be configured between very local (the most likely recipient is close along the ring) and entirely random (any recipient is equally likely).

Temporally, communication can be configured continuously, between a constant delay and a zero lookahead exponentially distributed wait time. The local RNG state is also used to generate the new event timestamp. A load imbalance is also configurable, which modifies the variance of the number of events processed at an entity's location. The load is normally distributed, an initial choice supported by the literature [250].

The performance metric of interest is the mean event processing rate across the entire simulation execution time, i.e., the number of events finally processed per unit of wall-clock time. The model itself is initialized in a steady state. Therefore,

Name	Range	Description
n_{entities}	$[1, \infty)$	Number of entities in the system
locality	$[0, 1)$	The lower the locality is set, the further along the ring topology messages are sent
load variance	$[0, \infty)$	The variance of the total number of events processed (sent and received) at one entity
$n_{\text{events\_per\_entity}}$	$[1, \infty)$	The total number of events in the system normalized to the number of entities n_{entities}
$r_{\text{ratio\_constant\_offset}}$	$(0, 1)$	Ratio of the constant offset (i.e., lookahead) as part of the average scheduling delay for an event
$r_{\text{internal\_events}}$	$[0, 1]$	Percentage of events that are not sent to another entity but rescheduled at the same entity
memory per entity	$[1, \infty)$	Memory (in chunks of 64 bit) per entity
memory per event	$[1, \infty)$	Memory (in chunks of 64 bit) per event and entity
seed	64 bit	Seed for the initial state
extra compute load	$[0, \infty)$	Amount of additional computations when handling an event
t_{end}	$(0, \infty)$	Logical termination time of the simulation

Table 1: Parameters of the configurable PDES benchmark model

an increase in the logical end time does not change the average computation load on the system other than what is incurred by simulator overhead.

Determinism under Simultaneous Events

One methodological challenge we had to approach was determinism under simultaneous events. As discussed in Section 2.4, this is one of the challenges, even with double-precision floating point numbers. To achieve determinism under the constraints of different simulators, we handle simultaneous events on the model side.

The key idea is to only advance an entity's state on an event with a timestamp that is guaranteed to be unique. This is achieved by only finalizing a state transition triggered at logical time t once the entity has advanced to an event at $t' > t$. When first encountering an event with a larger timestamp than the previous event, the entity stores its current state before advancing the state. If another event with the same timestamp needs to be handled, the previous state is restored. As the entity state sent to other entities is delayed by one event, an entity never sends a message based on a state change that will later be undone. Further considerations are made to ensure each message sent when processing simultaneous events is unique by basing messages on the previous entity state and a simultaneous event counter. This state-saving mechanism is part of the model and thus operates independently of the simulator implementation. On the model side, never more than one previous state needs to be saved.

As an alternative to storing the previous entity state, model-side reverse computation could be applied: in this variant, the last received event would be stored

	high local compute	*high memory*	*high queue load*	*small phold*
optimistic tool I	**23,000 ± 1,000**	**105,000 ± 2,000**	1,070	2,304,000 ± 11,000
optimistic tool II	**24,530 ± 40**	31,850 ± 80	**6,714,000 ± 14,000**	4,028,000 ± 9,000
conservative tool	9,328 ± 17	36,910 ± 70	135,100 ± 200	86,400 ± 700
sequential tool	5,463 ± 3	35,230 ± 30	2,184,300 ± 900	**5,220,600 ± 1,300**

Table 2: Performance (events/second) of differently configured models in events per second across different (anonymized) tools. The fastest is marked in **bold**. We can see the significant differences between the different tools for these scenarios on one particular technical setup (single node).

to be able to reverse its effect on the state when another event with the same timestamp is executed.

Results The differences in the tools' throughput were very high. In Table 2, we have four different configurations of the benchmarks and can see how four different tools have particularly strong or weak results at different configurations. This naturally leads to the combination of different simulators to achieve optimal performance across different workloads, which has been a recent trend in PDES [253,254].

5.3 Benchmarking Stochastic Simulation Algorithms

In the previous section, we devised a deterministic configurable benchmark model for Parallel Discrete Event Simulation (PDES). The motivation to introduce a benchmark for Stochastic Simulation Algorithms (SSA) and its many variations and implementations, as discussed in Section 3.1, is very similar. Therefore, we have also worked towards a comparable benchmarking study among them. The challenges here were more technical than conceptual. We build on existing work, such as a study on network-based vs. network-free approaches by Gupta et al. [71]. Jeschke et al. investigated the design space of SSA implementations using a component-based benchmark study [255].

No determinism Some challenges here differ from those of PDES. First of all, determinism is not achievable in the context of SSA, as each simulation algorithm uses different implementations of different algorithms with different random number generators. Specifically, the random process used for reaction selection differs across implementations and could not be part of the model the way we did it above. These algorithms will result in valid but different samples of the continuous-time Markov Chain. Furthermore, approximative algorithms like τ-leaping [76] are far more prevalent in SSA. Therefore, comparisons are always stochastic. The goal, instead, is to find a benchmark that suitably represents real-life workloads. Deterministic simulation algorithms, i.e., interpreting populations as concentrations leading to coupled ordinary differential equations (ODEs), have been evaluated regarding their performance [256]. However, the challenges are very different there because ODE integrators differ in their exactness.

Tool landscape There are more tools currently under active development in this domain than in PDES. Interfacing with these was a challenge. Ideally, all tools would support a standardized description of the simulation models and associated experiments. A recent advance in this field was BioSimulators [257], which is a broad frame-

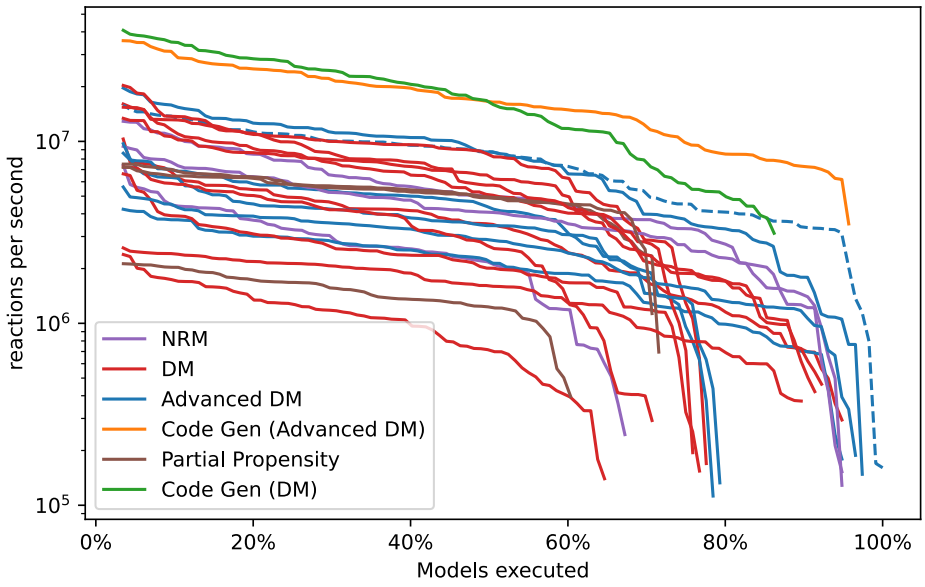


Figure 6: Different execution speeds across various models from our database for different simulation tools. The line indicates the percentage of models that can be executed at at least that speed. Different methods correspond to different line colors. The tree-odm (Section 3.4) method, is an advanced Direct Method (DM), but marked specifically with a dashed line here.

work for biological simulation tools. However, we found that the supplied docker images did not work for all simulators, requiring manual creation of these interfaces, utilizing in part existing specification languages and tools like SBML. This work was conducted by a Herrmann for a total of 10 simulation tools [258]. Now, we have the means to benchmark the execution of simple mass-action kinetic simulation models for Copasi [81], tellurium [259], GillesPy2 [260], rebob [261], pSSAlib [262], StochKit2 [263], and our own tools ML-Rules 2 [199] and ML-Rules 3 [E]. Finally, we included a custom simulation tool that has no additional features other than optimized implementations of basic simulation algorithms, which we call MinimalSSA. Many of these tools also support multiple simulation algorithms.

Benchmark models

One major advantage of the field of SSA for benchmarking is the plethora of available models, which can be executed by different tools. With some technical effort, we were able to automate this process. We extracted models from a previous benchmarking study [71], the *BioModels-Database* [264], and the *JWS Online model database* [265]. We removed all models that were not executable as mass action SSA. We also did further filtering. For example, we removed duplicates or removed those that where not executable. In total, we have 116 models. These models now serve as a foundation for a benchmark.

In Figure 6, we see the (filtered) simulation speed across the various tools and their algorithms for the models. These benchmarks are automated following the principles described by Ewald [266]. For each simulator, the models are sorted in decreasing order of performance. Therefore, the same x-axis point may not correspond to the same model across simulators. Some simulators drop off faster

than others, indicating a worse scaling. We can see the exceptional performance of the code generation methods (Section 3.5). We also observe that many methods struggle to scale beyond 60-80% of models. This mostly affects those implementations that do not use dependency tracking. Nonetheless, we find that the quality of implementation far outweighs the choice of algorithm for most cases. While not the fastest for small models, we find that our tree-odm (Section 3.4) implementations, with and without simulator specialization through code generation, scale very well for hard-to-execute models.

Data analysis

Using this data set, we now have many opportunities for further investigation into the data. For example, we can now tune our simulators for *typical* system sizes and structures. Based on this data set, we can also design a synthetic benchmark model similar to what was done in the previous section. We also find that the difference in performance is less model-dependent in SSA than it is in PDES. A fast tool will generally be fast across a wider range of models. The exceptions here are very basic algorithms (i.e., those without the use of dependency graphs as described in Section 3.2). Those might be fast for small models, but performance drops for larger ones.

We also see several data science and visualization opportunities based on this dataset. For example, we have explored how different model characteristics result in different execution performance. It would also be interesting to reduce the model set to a representative subset, which could be used to assess a simulation tool along core metrics quickly.

Model generation

Finally, it is a logical next step to explore the creation of synthetic models with specific performance properties to evaluate simulation tools. Using our dataset of real-world models, we can build on the state of the art [267,268]. We have explored the creation of synthetic models using genetic algorithms and geometric patterns. The model corpus from above could be used to validate that the synthetic models were representative of real-world scenarios, and we could sufficiently represent the performance characteristics of more than 80% of the models.

5.4 Summary

Reproducing research is a core concept of science. This chapter started with a brief discussion of the reproducibility crisis and potential remedies against it. In the following, we proposed two benchmarking solutions for different domains that address the challenges of evaluation and representativity of research on efficient stochastic simulation.

6 Conclusion

In the previous five chapters, we have identified the ingredients for *efficient abstraction and execution of stochastic simulation models*. In Chapter 1, we discussed the need for efficiency in simulation. In particular, we outlined the different types of efficiency as they pertain to the simulation process. We saw how metrics of efficiency could lead to different choices in hardware, like GPUs. In Chapter 2, we delved into stochasticity as an abstraction in general and its specific application in cellular biology. Here, we structured the space of simulation methods along different criteria. We found that for the domain of our choice, we could not operate on first principles alone and needed to add artificial reactions as a core concept. We also had our first performance case study analyzing and improving the implementation of a probabilistic cellular automaton.

The cell biological background of this chapter and the event-driven simulation concept introduced towards its end were combined in Chapter 3. The resulting Continuous-time Markov Chain approaches were covered in different applications throughout this thesis. After an introduction to model interfaces, with a focus on domain-specific languages, we gave some background on their efficient execution. This included work on their parallel execution, optimized versions of algorithms, and new optimization strategies in the form of simulator specialization. We expanded on the reaction concept by introducing dynamic structure models in Chapter 4. These dynamic structures allowed us to create variations for cell biological systems with dynamic hierarchy structures, both in an individual-based spatial approach and in a population-based approach. We applied what we learned in domain-specific languages and efficient simulation to agent-based simulation, resulting in an efficient implementation of a concise embedded modeling language for demography.

Finally, in Chapter 5, we discussed the issue of replicating and reproducing performance results, especially within the context of stochastic simulation. Here, we proposed two approaches tailored to different domains and their intricacies. For parallel discrete event simulation, we created a synthetic benchmark model, which identified the strengths and weaknesses when comparing different simulations. For reaction network simulators, we gathered benchmark models from different repositories. Here, we saw how different implementations of the same method resulted in very different execution speeds.

We have explored various advancements in efficient simulation, paving the way for simulating models on larger scales, exploring broader ranges of model parameters, and identifying suitable tools. We have successfully combined accessible and fast simulation techniques and addressed challenges with domain-specific languages. These outcomes rest on a solid foundation of well-defined physical principles, resulting in methods that are both sound and reproducible.

The reader is now prepared for the six publications that make up the core of this thesis, which are listed below. Naturally, these publications have opened up many potential future avenues of research, which are given as part of each individual publication.

Bibliography

[1] T. Köster, T. Warnke, and A. M. Uhrmacher, *Partial Evaluation via Code Generation for Static Stochastic Reaction Network Models*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS 2020)* (ACM, New York, USA, 2020), pp. 159–170

[2] T. Köster, K. Perumalla, and A. M. Uhrmacher, *Efficient Simulation of Nested Hollow Sphere Intersections for Dynamically Nested Compartmental Models in Cell Biology*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS 2017)* (ACM New York, NY, USA, 2017), pp. 173–183

[3] T. Köster and A. M. Uhrmacher, *Handling Dynamic Sets of Reactions in Stochastic Simulation Algorithms*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS 2018)* (ACM, New York, NY, USA, 2018), pp. 161–164

[4] T. Köster and A. M. Uhrmacher, *Efficient Execution for Domain Specific Languages: Comparing Two Approaches for Demography and Cellular Biology*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Association for Computing Machinery (ACM), New York, NY, USA, 2023), pp. 46–47

[5] T. Köster, A. M. Uhrmacher, and P. Andelfinger, *Towards an Open Repository for Reproducible Performance Comparison of Parallel and Distributed Discrete-Event Simulators*, in *SIGSIM Conference on Principles of Advanced Discrete Simulation* (ACM, New York, NY, USA, 2022), pp. 31–32

[6] T. Köster, *Efficient Simulation of Cell-Biological Multi-Level Models*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS 2019)* (2019)

[7] T. Köster, N. M. Drüeke, and A. M. Uhrmacher, *Latency Optimized Execution of Sequential Simulators by Parallel Parameter Optimization*, in *Winter Simulation Conference (WSC 2018)* (2018)

[8] T. Köster, F. Hauptmann, and A. M. Uhrmacher, *Optimizing Data Structures for Highly Dynamic Content in Collective, Adaptive Systems*, in *Winter Simulation Conference (WSC 2018)* (2018)

[9] T. Köster and A. M. Uhrmacher, *Multi-Level Particle-Based Modeling and Simulation of Cell Biological Systems*, in *V International Conference on Particle-Based Methods* (2017)

[10] T. Köster, *Reproducibility Report for the Paper: Workload Interference Prevention with Intelligent Routing and Flexible Job Placement on Dragonfly*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Association for Computing Machinery (ACM), New York, NY, USA, 2023), pp. 151–153

[11] T. Köster, *Reproducibility Report for the Paper: Spatial/temporal Locality-Based Load-Sharing in Speculative Discrete Event Simulation on Multi-Core Machines*, in *SIGSIM Conference on Principles of Advanced Discrete Simulation* (ACM, New York, NY, USA, 2022), pp. 134–137

[12] T. Köster, *Reproducibility Report for the Paper: Speculative Distributed Simulation of Very Large Spiking Neural Networks*, in *SIGSIM Conference on Principles of Advanced Discrete Simulation* (ACM, New York, NY, USA, 2022), pp. 141–144

[13] P. Henning, T. Köster, F. Haack, K. Burrage, and A. M. Uhrmacher, *Implications of Different Membrane Compartmentalization Models in Particle-Based in Silico Studies*, Royal Society Open Science **10**, 221177 (2023)

[14] F. Haack, T. Köster, and A. M. Uhrmacher, *Receptor/raft Ratio Is a Determinant for Lrp6 Phosphorylation and WNT/β-Catenin Signaling*, Frontiers in Cell and Developmental Biology **9**, 2085 (2021)

[15] P. Andelfinger, T. Köster, and A. M. Uhrmacher, *Zero Lookahead? Zero Problem. The Window Racer Algorithm*, in *SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS 2023)* (Association for Computing Machinery (ACM), New York, NY, USA, 2023), pp. 1–11

[16] J. Li, T. Köster, and P. J. Giabbanelli, *Design and Evaluation of Update Schemes to Optimize Asynchronous Cellular Automata with Random or Cyclic Orders*, in *The 25th International Symposium on Distributed Simulation and Real Time Applocations (IEEE/ACM DS-RT 2021)* (IEEE Computer Society, Los Alamitos, CA, USA, 2021), pp. 1–8

[17] J. N. Kreikemeyer, T. Köster, A. M. Uhrmacher, and T. Warnke, *Inferring Dependency Graphs for Agent-Based Models Using Aspect-Oriented Programming*, in *Winter Simulation Conference (WSC 2021)* (IEEE Press, 2021), pp. 1–12

[18] P. J. Giabbanelli, T. Köster, J. A. Devita, and J. A. Kohrt, *Optimizing Discrete Simulations of the Spread of HIV-1 to Handle Billions of Cells on Aworkstation*, in *ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS 2020)* (ACM, New York, USA, 2020), pp. 67–78

[19] F. J. Müller, F. Haack, T. Koester, and A. M. Uhrmacher, *Computational Model of Cell Cell Interaction in Bone Remodelling*, in *ELAINE 2024* (2024)

[20] P. Henning, T. Köster, and A. M. Uhrmacher, *Challenges in Reproducing an ODE Vesicle Transport Model with Particles in Continuous Space*, in *Multiscale Modeling and Simulations to Bridge Molecular and Cellular Scales* (2018)

[21] P. Henning, T. Köster, and A. M. Uhrmacher, *Idiosyncracies in Multi-Spatial Modeling*, in *15th Conference on Computational Methods in Systems Biology (CMSB 2017)* (2017)

[22] D. Goldsman, R. E. Nance, and J. R. Wilson, *A Brief History of Simulation Revisited*, in *Proceedings of the 2010 Winter Simulation Conference* (2010), pp. 567–574

[23] K. Nygaard and O.-J. Dahl, *The Development of the SIMULA Languages*, History of Programming Languages 439 (1978)

[24] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming* (Addison-Wesley Professional, 2010)

[25] A. Paszke et al., *Pytorch: An Imperative Style, High-Performance Deep Learning Library*, Advances in Neural Information Processing Systems **32**, (2019)

[26] L. Dematté and D. Prandi, *GPU Computing for Systems Biology*, Briefings in Bioinformatics **11**, 323 (2010)

[27] F. E. Cellier and J. Greifeneder, *Continuous System Modeling* (Springer Science & Business Media, 2013)

[28] D. Edwards and M. Hamson, *Mathematical Modelling* (Macmillan Press Ltd, London, UK, 1989)

[29] M. Pinsky and S. Karlin, *An Introduction to Stochastic Modeling* (Academic press, 2010)

[30] D. J. Wilkinson, *Stochastic Modelling for Systems Biology* (CRC press, 2018)

[31] M. Binois, J. Huang, R. B. Gramacy, and M. Ludkovski, *Replication or Exploration? Sequential Design for Stochastic Simulation Experiments*, Technometrics **61**, 7 (2019)

[32] R. M. Fujimoto, M. Hunter, A. Biswas, M. Jackson, and S. Neal, *Power Efficient Distributed Simulation*, in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2017), pp. 77–88

[33] D. H. Kim, C. Imes, and H. Hoffmann, *Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics*, in *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, And Applications* (2015), pp. 78–85

[34] A. Das, G. V. Merrett, and B. M. Al-Hashimi, *The Slowdown or Race-to-Idle Question: Workload-Aware Energy Optimization of SMT Multicore Platforms under Process Variation*, in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2016), pp. 535–538

[35] S. Conoci, M. Ianni, R. Marotta, and A. Pellegrini, *Autonomic Power Management in Speculative Simulation Runtime Environments*, in *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2020), pp. 93–98

[36] J. Himmelspach, R. Ewald, S. Leye, and A. M. Uhrmacher, *Enhancing the Scalability of Simulations by Embracing Multiple Levels of Parallelization*, in *2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, And Second International Workshop on High Performance Computational Systems Biology* (2010), pp. 57–66

[37] J. Shalf, *The Future of Computing Beyond Moore's Law*, Philosophical Transactions of the Royal Society a **378**, 20190061 (2020)

[38] G. Chrysos, *Intel® Xeon Phi™ Coprocessor-the Architecture*, Intel Whitepaper **176**, 43 (2014)

[39] K. O. W. Group and others, *The Opencl Specification Version 1.1*, Http://www. Khronos. Org/registry/cl/specs/opencl-1.1. Pdf (2011)

[40] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with Gpus*, (2012)

[41] H. Park and P. A. Fishwick, *A GPU-Based Application Framework Supporting Fast Discrete-Event Simulation*, Simulation **86**, 613 (2010)

[42] Y. Zhu, B. Wang, and Y. Deng, *Massively Parallel Logic Simulation with Gpus*, ACM Transactions on Design Automation of Electronic Systems (TODAES) **16**, 1 (2011)

[43] J. Xiao, P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll, *A Survey on Agent-Based Simulation Using Hardware Accelerators*, ACM Computing Surveys (CSUR) **51**, 1 (2019)

[44] T. Köster, Combined Particle and Compartmental Dynamics of Cell-Biological Models Using Hollow Spheres on the GPU, 2017

[45] P. A. Fishwick, *The Role of Process Abstraction in Simulation*, IEEE Transactions on Systems, Man, And Cybernetics **18**, 18 (1988)

[46] R. J. Brooks and A. M. Tobias, *Choosing the Best Model: Level of Detail, Complexity, And Model Performance*, Mathematical and Computer Modelling **24**, 1 (1996)

[47] R. B. Cooper, *Queueing Theory*, in *Proceedings of the Acm'81 Conference* (1981), pp. 119–122

[48] C. S. Pillay, J.-H. Hofmeyr, L. N. Mashamaite, and J. M. Rohwer, *From Top-down to Bottom-up: Computational Modeling Approaches for Cellular Redoxin Networks*, Antioxidants & Redox Signaling **18**, 2075 (2013)

[49] J. Moller and J. J. de Pablo, *Bottom-up Meets Top-down: The Crossroads of Multiscale Chromatin Modeling*, Biophysical Journal **118**, 2057 (2020)

[50] T. Székely Jr and K. Burrage, *Stochastic Simulation in Systems Biology*, Computational and Structural Biotechnology Journal **12**, 14 (2014)

[51] W. Lim, B. Mayer, and T. Pawson, *Cell Signaling* (Garland Science, 2014)

[52] F. J. Hartmann et al., *Single-Cell Metabolic Profiling of Human Cytotoxic t Cells*, Nature Biotechnology **39**, 186 (2021)

[53] H.-Y. Chuang, M. Hofree, and T. Ideker, *A Decade of Systems Biology*, Annual Review of Cell and Developmental Biology **26**, 721 (2010)

[54] M. Tomita, *Whole-Cell Simulation: A Grand Challenge of the 21st Century*, TRENDS in Biotechnology **19**, 205 (2001)

[55] U. Wolff, *Numerical Simulation in Quantum Field Theory*, Computational Physics: Selected Methods Simple Exercises Serious Applications 245 (1996)

[56] R. L. Delgado, S. Steinbeißer, M. Strickland, and J. H. Weber, *The Relativistic Schrödinger Equation Through FFTW 3: An Extension of Quantumfdtd*, Computer Physics Communications **272**, 108250 (2022)

[57] D. C. Rapaport, *The Art of Molecular Dynamics Simulation*, 2nd ed. (Cambridge University Press, Cambridge, UK ; New York, NY, 2004)

[58] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. Berendsen, *GROMACS: Fast, Flexible, And Free*, Journal of Computational Chemistry **26**, 1701 (2005)

[59] A. P. Thompson et al., *LAMMPS-a Flexible Simulation Tool for Particle-Based Materials Modeling at the Atomic, Meso, And Continuum Scales*, Computer Physics Communications **271**, 108171 (2022)

[60] A. Szabo and N. S. Ostlund, *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*, New ed. (Dover Publications Inc., Mineola, N.Y., 1996)

[61] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, *Scalable Molecular Dynamics with NAMD*, Journal of Computational Chemistry **26**, 1781 (2005)

[62] R. M. Venable, A. Kramer, and R. W. Pastor, *Molecular Dynamics Simulations of Membrane Permeability*, Chemical Reviews **119**, 5954 (2019)

[63] M. Arnittali, A. N. Rissanou, and V. Harmandaris, *Structure of Biomolecules Through Molecular Dynamics Simulations*, Procedia Computer Science **156**, 69 (2019)

[64] D. L. Ermak and H. Buckholz, *Numerical Integration of the Langevin Equation: Monte Carlo Simulation*, Journal of Computational Physics **35**, 169 (1980)

[65] J. Tóthová and V. Lisỳ, *Overdamped and Underdamped Langevin Equations in the Interpretation of Experiments and Simulations*, European Journal of Physics **43**, 65103 (2022)

[66] P. Dittrich, J. Ziegler, and W. Banzhaf, *Artificial Chemistries–a Review*, Artificial Life **7**, 225 (2001)

[67] H. Fellermann, *Spatially Resolved Artificial Chemistry*, Artificial Life Models in Software 343 (2009)

[68] W. Banzhaf and L. Yamamoto, *Artificial Chemistries* (MIT Press, 2024)

[69] M. E. Johnson, A. Chen, J. R. Faeder, P. Henning, I. I. Moraru, M. Meier-Schellersheim, R. F. Murphy, T. Prüstel, J. A. Theriot, and A. M. Uhrmacher, *Quantifying the Roles of Space and Stochasticity in Computer Simulations for Cell Biology and Cellular Biochemistry*, Molecular Biology of the Cell **32**, 186 (2021)

[70] K. Takahashi, S. N. V. Arjunan, and M. Tomita, *Space in Systems Biology of Signaling Pathways - Towards Intracellular Molecular Crowding in Silico*, FEBS Letters **579**, 1783 (2005)

[71] A. Gupta and P. Mendes, *An Overview of Network-Based and -Free Approaches for Stochastic Simulation of Biochemical Systems*, Computation **6**, (2018)

[72] D. T. Gillespie, *Stochastic Simulation of Chemical Kinetics*, Annu. Rev. Phys. Chem. **58**, 35 (2007)

[73] D. T. Gillespie, *A Rigorous Derivation of the Chemical Master Equation*, Physica A: Statistical Mechanics and Its Applications **188**, 404 (1992)

[74] W. J. Anderson, *Continuous-Time Markov Chains: An Applications-Oriented Approach* (Springer Science & Business Media, 2012)

[75] D. T. Gillespie, *Exact Stochastic Simulation of Coupled Chemical Reactions*, The Journal of Physical Chemistry **81**, 2340 (1977)

[76] D. T. Gillespie, *Approximate Accelerated Stochastic Simulation of Chemically Reacting Systems*, The Journal of Chemical Physics **115**, 1716 (2001)

[77] K. Burrage, T. Tian, and P. Burrage, *A Multi-Scaled Approach for Simulating Chemical Reaction Systems*, Progress in Biophysics and Molecular Biology **85**, 217 (2004)

[78] D. Higham and P. Kloeden, *An Introduction to the Numerical Simulation of Stochastic Differential Equations* (SIAM, 2021)

[79] K. Burrage, P. Burrage, and T. Tian, *Numerical Methods for Strong Solutions of Stochastic Differential Equations: An Overview*, Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences **460**, 373 (2004)

[80] D. Adalsteinsson, D. McMillen, and T. C. Elston, *Biochemical Network Stochastic Simulator (Bionets): Software for Stochastic Modeling of Biochemical Networks*, BMC Bioinformatics **5**, 1 (2004)

[81] S. Hoops, S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer, *COPASI —a Complex Pathway Simulator*, Bioinformatics **22**, 3067 (2006)

[82] S. S. Andrews, *Smoldyn: Particle-Based Simulation with Rule-Based Modeling, Improved Molecular Interaction and a Library Interface*, Bioinformatics **33**, 710 (2017)

[83] A. Regev, E. M. Panina, W. Silverman, L. Cardelli, and E. Shapiro, *BioAmbients: An Abstraction for Biological Compartments*, Theoretical Computer Science **325**, 141 (2004)

[84] R. A. Kerr, T. M. Bartol, B. Kaminsky, M. Dittrich, J.-C. J. Chang, S. B. Baden, T. J. Sejnowski, and J. R. Stiles, *Fast Monte Carlo Simulation Methods for Biological Reaction-Diffusion Systems in Solution and on Surfaces*, SIAM Journal on Scientific Computing **30**, 3126 (2008)

[85] S. S. Andrews, N. J. Addy, R. Brent, and A. P. Arkin, *Detailed Simulations of Cell Biology with Smoldyn 2.1*, Plos Computational Biology **6**, 1000705 (2010)

[86] M. L. Blinov, J. C. Schaff, D. Vasilescu, I. I. Moraru, J. E. Bloom, and L. M. Loew, *Compartmental and Spatial Rule-Based Modeling with Virtual Cell*, Biophysical Journal **113**, 1365 (2017)

[87] J. Hattne, D. Fange, and J. Elf, *Stochastic Reaction-Diffusion Simulation with MesoRD*, Bioinformatics **21**, 2923 (2005)

[88] J. Schöneberg, A. Ullrich, and F. Noé, *Simulation Tools for Particle-Based Reaction-Diffusion Dynamics in Continuous Space*, BMC Biophysics **7**, 11 (2014)

[89] P. J. Michalski and L. M. Loew, *SpringSaLaD: A Spatial, Particle-Based Biochemical Simulation Platform with Excluded Volume*, Biophysical Journal **110**, 523 (2016)

[90] J. Schöneberg and F. Noé, *ReaDDy - A Software for Particle-Based Reaction-Diffusion Dynamics in Crowded Cellular Environments*, Plos ONE **8**, 74261 (2013)

[91] G. Gruenert, B. Ibrahim, T. Lenser, M. Lohel, T. Hinze, and P. Dittrich, *Rule-Based Spatial Modeling with Diffusing, Geometrically Constrained Molecules*, BMC Bioinformatics **11**, 307 (2010)

[92] R. M. Donovan, J.-J. Tapia, D. P. Sullivan, J. R. Faeder, R. F. Murphy, M. Dittrich, and D. M. Zuckerman, *Unbiased Rare Event Sampling in Spatial Stochastic Systems Biology Models Using a Weighted Ensemble of Trajectories*, PLOS Computational Biology **12**, 1004611 (2016)

[93] A. T. Bittig and A. M. Uhrmacher, *ML-Space: Hybrid Spatial Gillespie and Particle Simulation of Multi-Level Rule-Based Models in Cell Biology*, IEEE/ACM Transactions on Computational Biology and Bioinformatics **14**, 1339 (2017)

[94] M. B. Flegg, S. J. Chapman, and R. Erban, *The Two-Regime Method for Optimizing Stochastic Reaction-Diffusion Simulations*, Journal of the Royal Society, Interface **9**, 859 (2012)

[95] M. Klann, A. Ganguly, and H. Koeppl, *Hybrid Spatial Gillespie and Particle Tracking Simulation*, Bioinformatics **28**, (2012)

[96] Y. Kim, M. A. Stolarska, and H. G. Othmer, *A Hybrid Model for Tumor Spheroid Growth in Vitro I: Theoretical Development and Early Results*, Mathematical Models and Methods in Applied Sciences **17**, 1773 (2007)

[97] J. C. Schaff, F. Gao, Y. Li, I. L. Novak, and B. M. Slepchenko, *Numerical Approach to Spatial Deterministic-Stochastic Models Arising in Cell Biology*, PLOS Computational Biology **12**, 1005236 (2016)

[98] R. M. Z. dos Santos and S. Coutinho, *Dynamics of HIV Infection: A Cellular Automata Approach*, Physical Review Letters **87**, 168102 (2001)

[99] S. Wolfram, *Statistical Mechanics of Cellular Automata*, Reviews of Modern Physics **55**, 601 (1983)

[100] T. Sterling, M. Brodowicz, and M. Anderson, *High Performance Computing: Modern Systems and Practices* (Morgan Kaufmann, 2017)

[101] M. C. Strain and H. Levine, *Comment on "Dynamics of HIV Infection: A Cellular Automata Approach"*, Physical Review Letters **89**, 219805 (2002)

[102] B. Schönfisch and A. de Roos, *Synchronous and Asynchronous Updating in Cellular Automata*, in *Cellular Automata: Research Towards Industry*, edited by S. Bandini, R. Serra, and F. S. Liverani (Springer London, London, 1998), pp. 42–46

[103] J. Metzcar, Y. Wang, R. Heiland, and P. Macklin, *A Review of Cell-Based Computational Modeling in Cancer Biology*, JCO Clinical Cancer Informatics 1 (2019)

[104] B. Schönfisch, *Propagation of Fronts in Cellular Automata*, Physica D: Nonlinear Phenomena **80**, 433 (1995)

[105] A. M. Law, *Simulation Modeling & Analysis*, 4th ed. (McGraw-Hill, New York, NY, USA, 2007)

[106] R. M. Fujimoto, *Parallel Discrete Event Simulation*, Communications of the ACM **33**, 30 (1990)

[107] P. Andelfinger and W. Cai, *Advanced Tutorial: Parallel and Distributed Methods for Scalable Discrete Simulation*, in *Proceedings of the Winter Simulation Conference (WSC)* (2022)

[108] P. Peschlow and P. Martini, *A Discrete-Event Simulation Tool for the Analysis of Simultaneous Events*, in *1st International ICST Workshop on Network Simulation Tools* (2010)

[109] D. Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys (CSUR) **23**, 5 (1991)

[110] A. DasGupta, *The Matching, Birthday and the Strong Birthday Problem: A Contemporary Review*, Journal of Statistical Planning and Inference **130**, 377 (2005)

[111] R. Ronngren and M. Liljenstam, *On Event Ordering in Parallel Discrete Event Simulation*, in *Proceedings Thirteenth Workshop on Parallel and Distributed Simulation. PADS 99.(Cat. No. Pr00155)* (1999), pp. 38–45

[112] A. Piccione and A. Pellegrini, *Practical Tie-Breaking for Parallel/distributed Simulations*, in *2023 IEEE/ACM 27th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (2023), pp. 74–83

[113] B. P. Zeigler, *Discrete Event Models for Cell Space Simulation*, International Journal of Theoretical Physics **21**, 573 (1982)

[114] G. A. Wainer, *Cellular Modeling with Cell-DEVS: A Discrete-Event Cellular Automata Formalism*, in *Cellular Automata* (Springer International Publishing, Cham, 2014), pp. 6–15

[115] P. G. Fennell, S. Melnik, and J. P. Gleeson, *Limitations of Discrete-Time Approaches to Continuous-Time Contagion Dynamics*, Physical Review E **94**, 52125 (2016)

[116] S. Tisue and U. Wilensky, *Netlogo: A Simple Environment for Modeling Complexity*, in *International Conference on Complex Systems*, Vol. 21 (2004), pp. 16–21

[117] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems* (Academic press, 2000)

[118] S. Zschaler and F. A. Polack, *Trustworthy Agent-Based Simulation: The Case for Domain-Specific Modelling Languages*, Software and Systems Modeling **22**, 455 (2023)

[119] J. Kramer, *Is Abstraction the Key to Computing?*, Communications of the ACM **50**, 36 (2007)

[120] T. Warnke, Domain-Specific Languages for Modeling and Simulation, 2021

[121] W. K. Adams, S. Reid, R. LeMaster, S. McKagan, K. Perkins, M. Dubson, and C. E. Wieman, *A Study of Educational Simulations Part II–Interface Design*, Journal of Interactive Learning Research **19**, 551 (2008)

[122] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, in *Ecoop'93—Object-Oriented Programming: 7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7* (1993), pp. 406–431

[123] C. D. Carothers, D. Bauer, and S. Pearce, *ROSS: A High-Performance, Low-Memory, Modular Time Warp System*, Journal of Parallel and Distributed Computing **62**, 1648 (2002)

[124] A. Pellegrini, R. Vitali, F. Quaglia, and others, *The Rome Optimistic Simulator (ROOT-Sim)*, (2013)

[125] J. Thompson, H. Zhao, S. Seneviratne, R. Byrne, R. Vidanaarachchi, and R. McClure, *A Template for Transfer of Netlogo Models to High-Performance Computing Environments for Enhanced Real-World Decision-Support*, in *Conference of the European Social Simulation Association* (2022), pp. 567–576

[126] S. Railsback, D. Ayllón, U. Berger, V. Grimm, S. Lytinen, C. Sheppard, and J. C. Thiele, *Improving Execution Speed of Models Implemented in Netlogo*, (2017)

[127] B. Geltz, *Handling External Events Efficiently in Gillespie's Stochastic Simulation Algorithm*, (2010)

[128] S. Matthew et al., *Gillespy2: A Biochemical Modeling Framework for Simulation Driven Biological Discovery*, Letters in Biomathematics **10**, 87 (2023)

[129] X. Cai, *Exact Stochastic Simulation of Coupled Chemical Reactions with Delays*, The Journal of Chemical Physics **126**, (2007)

[130] H. Ge and H. Qian, *Chemical Master Equation*, in *Encyclopedia of Systems Biology*, edited by W. Dubitzky, O. Wolkenhauer, K.-H. Cho, and H. Yokota (Springer New York, New York, NY, 2013), pp. 396–399

[131] L. Michaelis, M. L. Menten, and others, *Die Kinetik Der Invertinwirkung*, Biochem. Z **49**, 352 (1913)

[132] A. Cornish-Bowden, *One Hundred Years of Michaelis–Menten Kinetics*, Perspectives in Science **4**, 3 (2015)

[133] K. R. Sanft, D. T. Gillespie, and L. R. Petzold, *Legitimacy of the Stochastic Michaelis–Menten Approximation*, IET Systems Biology **5**, 58 (2011)

[134] A. V. Hill, *The Possible Effects of the Aggregation of the Molecules of Hemoglobin on Its Dissociation Curves*, J. Physiol. **40**, (1910)

[135] P. Smadbeck and Y. Kaznessis, *Stochastic Model Reduction Using a Modified Hill-Type Kinetic Rate Law*, The Journal of Chemical Physics **137**, (2012)

[136] Y. Cao, D. T. Gillespie, and L. R. Petzold, *The Slow-Scale Stochastic Simulation Algorithm*, The Journal of Chemical Physics **122**, (2005)

[137] D. A. Beard, *A Biophysical Model of the Mitochondrial Respiratory System and Oxidative Phosphorylation*, Plos Computational Biology **1**, 36 (2005)

[138] D. T. Gillespie, *A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions*, Journal of Computational Physics **22**, 403 (1976)

[139] M. A. Gibson and J. Bruck, *Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels*, The Journal of Physical Chemistry a **104**, 1876 (2000)

[140] Y. Cao, H. Li, and L. Petzold, *Efficient formulation of the stochastic simulation algorithm for chemically reacting systems*, The Journal of Chemical Physics **121**, 4059 (2004)

[141] J. M. McCollum, G. D. Peterson, C. D. Cox, M. L. Simpson, and N. F. Samatova, *The Sorting Direct Method for Stochastic Simulation of Biochemical Systems with Varying Reaction Execution Behavior*, Computational Biology and Chemistry **30**, 39 (2006)

[142] H. Li and L. Petzold, *Logarithmic Direct Method for Discrete Stochastic Simulation of Chemically Reacting Systems*, Journal of Chemical Physics **16**, 1 (2006)

[143] R. Ramaswamy and I. F. Sbalzarini, *A Partial-Propensity Variant of the Composition-Rejection Stochastic Simulation Algorithm for Chemical Reaction Networks*, The Journal of Chemical Physics **132**, (2010)

[144] Y. Cao, D. T. Gillespie, and L. R. Petzold, *Efficient Step Size Selection for the Tau-Leaping Simulation Method*, The Journal of Chemical Physics **124**, (2006)

[145] Y. Cao, D. T. Gillespie, and L. R. Petzold, *Avoiding Negative Populations in Explicit Poisson Tau-Leaping*, The Journal of Chemical Physics **123**, (2005)

[146] T. T. Marquez-Lago and K. Burrage, *Binomial Tau-Leap Spatial Stochastic Simulation Algorithm for Applications in Chemical Kinetics*, The Journal of Chemical Physics **127**, (2007)

[147] P. J. Andelfinger, *Identifying and Harnessing Concurrency for Parallel and Distributed Network Simulation* (KIT Scientific Publishing, 2016)

[148] K. Sumiyoshi, K. Hirata, N. Hiroi, and A. Funahashi, *Acceleration of Discrete Stochastic Biochemical Simulation Using GPGPU*, Frontiers in Physiology **6**, (2015)

[149] E. Kouskoumvekakis, D. Soudris, and E. S. Manolakos, *Many-Core Cpus Can Deliver Scalable Performance to Stochastic Simulations of Large-Scale Biochemical Reaction Networks*, in *2015 International Conference on High Performance Computing & Simulation (HPCS)* (2015), pp. 517–524

[150] V. H. Thanh and R. Zunino, *Parallel Stochastic Simulation of Biochemical Reaction Systems on Multi-Core Processors*, Proc. Of Cssim 162 (2011)

[151] C. S. Gillespie, *Stochastic Simulation of Chemically Reacting Systems Using Multi-Core Processors*, The Journal of Chemical Physics **136**, (2012)

[152] L. Macchiarulo, *A Massively Parallel Implementation of Gillespie Algorithm on Fpgas*, in *2008 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society* (2008), pp. 1343–1346

[153] G. Klingbeil, R. Erban, M. Giles, and P. K. Maini, *Fat Versus Thin Threading Approach on GPUs: Application to Stochastic Simulation of Chemical Reactions*, IEEE Transactions on Parallel and Distributed Systems **23**, 280 (2012)

[154] I. Komarov and R. M. D'Souza, *Accelerating the Gillespie Exact Stochastic Simulation Algorithm Using Hybrid Parallel Execution on Graphics Processing Units*, PLOS ONE **7**, 46693 (2012)

[155] G. Kunz, D. Schemmel, J. Gross, and K. Wehrle, *Multi-Level Parallelism for Time- and Cost-Efficient Parallel Discrete Event Simulation on Gpus*, in *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, Vol. 0 (2012), pp. 23–32

[156] D. Jenkins and G. Peterson, *Gpu Accelerated Stochastic Simulation*, in *Symposium on Application Accelerators in High Performance Computing (SAAHPC)* (2010)

[157] D. D. Jenkins and G. D. Peterson, *AESS: Accelerated Exact Stochastic Simulation*, Computer Physics Communications **182**, 2580 (2011)

[158] C. Dittamo and D. Cangelosi, *Optimized Parallel Implementation of Gillespie's First Reaction Method on Graphics Processing Units*, in *2009 International Conference on Computer Modeling and Simulation* (2009), pp. 156–161

[159] H. Li and L. Petzold, *Efficient Parallelization of the Stochastic Simulation Algorithm for Chemically Reacting Systems On the Graphics Processing Unit*, The International Journal of High Performance Computing Applications **24**, 107 (2009)

[160] G. Klingbeil, R. Erban, M. Giles, and P. K. Maini, *STOCHSIMGPU: Parallel Stochastic Simulation for the Systems Biology Toolbox 2 for MATLAB*, Bioinformatics **27**, 1170 (2011)

[161] J. Elf and M. Ehrenberg, *Spontaneous Separation of Bi-Stable Biochemical Systems into Spatial Domains of Opposite Phases*, Systems Biology **1**, 230 (2004)

[162] M. Jeschke, R. Ewald, A. Park, R. Fujimoto, and A. M. Uhrmacher, *A Parallel and Distributed Discrete Event Approach for Spatial Cell-Biological Simulations*, ACM SIGMETRICS Performance Evaluation Review **35**, 22 (2008)

[163] B. Wang, B. Hou, F. Xing, and Y. Yao, *Abstract Next Subvolume Method: A Logical Process-Based Approach for Spatial Stochastic Simulation of Chemical Reactions*, Computational Biology and Chemistry **35**, 193 (2011)

[164] B. Wang, Y. Yao, Y. Zhao, B. Hou, and S. Peng, *Experimental Analysis of Optimistic Synchronization Algorithms for Parallel Simulation of Reaction-Diffusion Systems*, in *2009 International Workshop on High Performance Computational Systems Biology* (2009), pp. 91–100

[165] P. Bauer, J. Lindén, S. Engblom, and B. Jonsson, *Efficient Inter-Process Synchronization for Parallel Discrete Event Simulation on Multicores*, in *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2015), pp. 183–194

[166] J. Linden, P. Bauer, S. Engblom, and B. Jonsson, *Exposing Inter-Process Information for Efficient Pdes of Spatial Stochastic Systems on Multicores*, ACM Transactions on Modeling and Computer Simulation (TOMACS) **29**, 1 (2019)

[167] M. Harris, S. Sengupta, and J. D. Owens, *Parallel Prefix Sum (Scan) with CUDA*, GPU Gems **3**, 851 (2007)

[168] H. Li and L. R. Petzold, *Logarithmic Direct Method for Discrete Stochastic Simulation of Chemically Reacting Systems*, in (2006)

[169] C. Versari and N. Busi, *Efficient Stochastic Simulation of Biological Systems with Multiple Variable Volumes*, Electronic Notes in Theoretical Computer Science **194**, 165 (2008)

[170] W. Kahan, *Pracniques: Further Remarks on Reducing Truncation Errors*, Communications of the ACM **8**, 40 (1965)

[171] P. Lee and M. Leone, *Optimizing ML with Run-Time Code Generation*, ACM SIGPLAN Notices **31**, 137 (1996)

[172] S. Erdweg et al., *The State of the Art in Language Workbenches: Conclusions from the Language Workbench Challenge*, in *Software Language Engineering: 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings 6* (2013), pp. 197–217

[173] T. Meyer, T. Helms, T. Warnke, and A. M. Uhrmacher, *Runtime Code Generation for Interpreted Domain-Specific Modeling Languages*, in *2018 Winter Simulation Conference (WSC)* (2018), pp. 605–615

[174] A. Kerr, G. Diamos, and S. Yalamanchili, *A Characterization and Analysis of Ptx Kernels*, in *2009 IEEE International Symposium on Workload Characterization (IISWC)* (2009), pp. 3–12

[175] C. Lattner and V. Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (2004), pp. 75–86

[176] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, *MLIR: Scaling Compiler Infrastructure for Domain Specific Computation*, in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2021), pp. 2–14

[177] E. T. Somogyi, J.-M. Bouteiller, J. A. Glazier, M. König, J. K. Medley, M. H. Swat, and H. M. Sauro, *Libroadrunner: A High Performance SBML Simulation and Analysis Library*, Bioinformatics **31**, 3315 (2015)

[178] Y. Futamura, *Partial Evaluation of Computation Process—AnApproach to a Compiler-Compiler*, Higher Order Symbol. Comput. **12**, 381 (1999)

[179] U. P. Schultz, J. L. Lawall, and C. Consel, *Automatic Program Specialization for Java*, ACM Transactions on Programming Languages and Systems **25**, 452 (2003)

[180] J. Zeng, Partial Evaluation for Code Generation from Domain-Specific Languages, Columbia University, 2007

[181] T. Rompf and M. Odersky, *Lightweight Modular Staging*, Communications of the ACM **55**, 121 (2012)

[182] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt, *Anydsl: A Partial Evaluation Framework for Programming High-Performance Libraries*, Proc. ACM Program. Lang. **2**, (2018)

[183] N. Segev, A. A. Tokarev, A. Alfonso, and N. Segev, *Overview of Intracellular Compartments and Trafficking Pathways*, Trafficking inside Cells: Pathways, Mechanisms and Regulation 3 (2009)

[184] L. A. Harris, J. S. Hogg, and J. R. Faeder, *Compartmental Rule-Based Modeling of Biochemical Systems*, in *Proceedings of the 2009 Winter Simulation Conference (WSC)* (2009), pp. 908–919

[185] J. Liu, Q. Xiao, J. Xiao, C. Niu, Y. Li, X. Zhang, Z. Zhou, G. Shu, and G. Yin, *Wntβ-Catenin Signalling: Function, Biological Mechanisms, And Therapeutic Opportunities*, Signal Transduction and Targeted Therapy **7**, 3 (2022)

[186] K. Budde, J. Smith, P. Wilsdorf, F. Haack, and A. M. Uhrmacher, *Relating Simulation Studies by Provenance–developing a Family of Wnt Signaling Models*, Plos Computational Biology **17**, 1 (2021)

[187] A. L. MacLean, Z. Rosen, H. M. Byrne, and H. A. Harrington, *Parameter-Free Methods Distinguish Wnt Pathway Models and Guide Design of Experiments*, Proceedings of the National Academy of Sciences **112**, 2652 (2015)

[188] F. Haack, H. Lemcke, R. Ewald, T. Rharass, and A. M. Uhrmacher, *Spatio-Temporal Model of Endogenous ROS and Raft-Dependent Wnt/beta-Catenin Signaling Driving Cell Fate Commitment in Human Neural Progenitor Cells*, PLOS Computational Biology **11**, 1 (2015)

[189] L. Harris, J. S. Hogg, and J. R. Faeder, *Compartmental Rule-Based Modeling of Biochemical Systems*, in *Proceedings of the 2009 Winter Simulation Conference* (IEEE, Austin, Texas, 2009)

[190] M. Hucka et al., *The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 2 Core Release 2*, Journal of Integrative Bioinformatics **16**, (2019)

[191] J. R. Faeder, M. L. Blinov, and W. S. Hlavacek, *Rule-Based Modeling of Biochemical Systems with Bionetgen*, Systems Biology 113 (2009)

[192] M. W. Sneddon, J. R. Faeder, and T. Emonet, *Efficient Modeling, Simulation and Coarse-Graining of Biological Complexity with Nfsim*, Nature Methods **8**, 177 (2011)

[193] C. Maus, S. Rybacki, and A. M. Uhrmacher, *Rule-Based Multi-Level Modeling of Cell Biological Systems*, BMC Systems Biology **5**, 166 (2011)

[194] T. Helms, T. Warnke, and A. M. Uhrmacher, *Multi-Level Modeling and Simulation of Cellular Systems: An Introduction to ML-Rules*, Modeling Biomolecular Site Dynamics: Methods and Protocols 141 (2019)

[195] T. Warnke, T. Helms, and A. M. Uhrmacher, *Syntax and Semantics of a Multi-Level Modeling Language*, in *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2015), pp. 133–144

[196] L. A. Harris, J. S. Hogg, J.-J. Tapia, J. A. P. Sekar, S. Gupta, I. Korsunsky, A. Arora, D. Barua, R. P. Sheehan, and J. R. Faeder, *BioNetGen 2.2: advances in rule-based modeling*, Bioinformatics **32**, 3366 (2016)

[197] P. Boutillier, M. Maasha, X. Li, H. F. Medina-Abarca, J. Krivine, J. Feret, I. Cristescu, A. G. Forbes, and W. Fontana, *The Kappa Platform for Rule-Based Modeling*, Bioinformatics **34**, (2018)

[198] C. D. Thompson-Walsh, J. Hayman, and G. Winskel, *Containment in Rule-Based Models*, Electronic Notes in Theoretical Computer Science **284**, 125 (2012)

[199] T. Helms, T. Warnke, C. Maus, and A. M. Uhrmacher, *Semantics and Efficient Simulation Algorithms of an Expressive Multi-Level Modeling Language*, ACM Transactions on Modeling and Computer Simulation **27**, 1 (2017)

[200] T. Helms, P. Wilsdorf, and A. M. Uhrmacher, *Hybrid Simulation of Dynamic Reaction Networks in Multi-Level Models*, in *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2018), pp. 133–144

[201] T. Helms, M. Luboschik, H. Schumann, and A. M. Uhrmacher, *An Approximate Execution of Rule-Based Multi-Level Models*, in *Computational Methods in Systems Biology: 11th International Conference, CMSB 2013, Klosterneuburg, Austria, September 22-24, 2013. Proceedings 11* (2013), pp. 19–32

[202] F. Baras and M. M. Mansour, *Reaction-Diffusion Master Equation: A Comparison with Microscopic Simulations*, Physical Review E **54**, 6139 (1996)

[203] A. T. Bittig, ML-Space: Hybrid Spatial Gillespie and Brownian Motion Simulation at Multiple Levels, And a Rule-Based Description Language, 2017

[204] E. M. Purcell, *Life at Low Reynolds Number*, in *Physics and Our World: Reissue of the Proceedings of a Symposium in Honor of Victor F Weisskopf* (2014), pp. 47–67

[205] W. T. Coffey, Y. P. Kalmykov, and J. T. Waldron, *The Langevin Equation: With Applications to Stochastic Problems in Physics, Chemistry and Electrical Engineering*, Revised. (World Scientific Pub Co Inc, Singapore; River Edge, N.J., 2004)

[206] T. Warnke, A. Steiniger, A. M. Uhrmacher, A. Klabunde, and F. Willekens, *Ml3: A Language for Compact Modeling of Linked Lives in Computational Demography*, in *2015 Winter Simulation Conference (WSC)* (2015), pp. 2764–2775

[207] R. Harper, *Programming in Standard ML* (2011)

[208] D. Hall and A. P. Minton, *Macromolecular Crowding: Qualitative and Semiquantitative Successes, Quantitative Challenges*, Biochimica Et Biophysica Acta (BBA) - Proteins and Proteomics **1649**, 127 (2003)

[209] J. Neefjes, M. M. L. Jongsma, and I. Berlin, *Stop or Go? Endosome Positioning in the Establishment of Compartment Architecture, Dynamics, And Function*, Trends in Cell Biology **27**, 580 (2017)

[210] R. Zunino, D. Nikolic, C. Priami, O. Kahramanogulları, and T. Schiavinotto, *L: An Imperative DSL to Stochastically Simulate Biological Systems*, Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday 354 (2015)

[211] C. M. Macal and M. J. North, *Tutorial on Agent-Based Modeling and Simulation*, in *Proceedings of the Winter Simulation Conference, 2005.* (2005), p. 14–pp

[212] T. Warnke, O. Reinhardt, A. Klabunde, F. Willekens, and A. M. Uhrmacher, *Modelling and Simulating Decision Processes of Linked Lives: An Approach Based on Concurrent Processes and Stochastic Race*, Population Studies **71**, 69 (2017)

[213] O. Reinhardt, T. Warnke, and A. M. Uhrmacher, *A Language for Agent-Based Discrete-Event Modeling and Simulation of Linked Lives*, ACM Transactions on Modeling and Computer Simulation (TOMACS) **32**, 1 (2022)

[214] T. Henzinger, B. Jobstmann, and V. Wolf, *Formalisms for Specifying Markovian Population Models*, International Journal of Foundations of Computer Science **22**, 823 (2011)

[215] P. Andelfinger, A. Piccione, A. Pellegrini, and A. Uhrmacher, *Comparing Speculative Synchronization Algorithms for Continuous-Time Agent-Based Simulations*, in *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (2022), pp. 57–66

[216] O. Reinhardt and A. M. Uhrmacher, *An Efficient Simulation Algorithm for Continuous-Time Agent-Based Linked Lives Models*, in *Proceedings of the 50th Annual Simulation Symposium* (2017), pp. 1–12

[217] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-Oriented Programming*, in *Ecoop'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11* (1997), pp. 220–242

[218] S. Luke, R. Simon, A. Crooks, H. Wang, E. Wei, D. Freelan, C. Spagnuolo, V. Scarano, G. Cordasco, and C. Cioffi-Revilla, *The MASON Simulation Toolkit: Past, Present, And Future*, in *Multi-Agent-Based Simulation XIX: 19th International Workshop, MABS 2018, Stockholm, Sweden, July 14, 2018, Revised Selected Papers 19* (2019), pp. 75–86

[219] M. Baker, *Reproducibility Crisis*, Nature **533**, 353 (2016)

[220] S. J. Taylor, T. Eldabi, T. Monks, M. Rabe, and A. M. Uhrmacher, *Crisis, What Crisis–Does Reproducibility in Modeling & Simulation Really Matter?*, in *2018 Winter Simulation Conference (WSC)* (2018), pp. 749–762

[221] O. Dalle, *On Reproducibility and Traceability of Simulations*, in *2012 Winter Simulation Conference* (2012), pp. 1–12

[222] I. Fister, J. Brest, A. Iglesias, A. Galvez, and S. Deb, *On Selection of a Benchmark by Determining the Algorithms' Qualities*, IEEE Access **9**, 51166 (2021)

[223] G. Cenikj, R. D. Lang, A. P. Engelbrecht, C. Doerr, P. Korošec, and T. Eftimov, *Selector: Selecting a Representative Benchmark Suite for Reproducible Statistical Comparison*, in *Proceedings of the Genetic and Evolutionary Computation Conference* (2022), pp. 620–629

[224] D. Saucez and L. Iannone, *Thoughts and Recommendations from the ACM SIGCOMM 2017 Reproducibility Workshop*, SIGCOMM Comput. Commun. Rev. **48**, 70 (2018)

[225] N. Ferro and D. Kelly, *SIGIR Initiative to Implement ACM Artifact Review and Badging*, SIGIR Forum **52**, 4 (2018)

[226] I. BiPM, I. IFCC, I. IUPAC, and O. ISO, *The International Vocabulary of Metrology—Basic and General Concepts and Associated Terms (VIM)*, Jcgm **200**, (2012)

[227] T. Monks, C. S. Currie, B. S. Onggo, S. Robinson, M. Kunc, and S. J. Taylor, *Strengthening the Reporting of Empirical Simulation Studies: Introducing the STRESS Guidelines*, Journal of Simulation **13**, 55 (2019)

[228] A. Fokkens, M. Van Erp, M. Postma, T. Pedersen, P. Vossen, and N. Freire, *Offspring from Reproduction Problems: What Replication Failure Teaches Us*, in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2013), pp. 1691–1701

[229] A. Ruscheinski and A. Uhrmacher, *Provenance in Modeling and Simulation Studies — Bridging Gaps*, in *2017 Winter Simulation Conference (WSC)*, Vol. 0 (2017), pp. 872–883

[230] P. Wilsdorf, O. Reinhardt, T. Prike, M. Hinsch, J. Bijak, and A. M. Uhrmacher, *Simulation Studies of Social Systems: Telling the Story Based on Provenance Patterns*, Royal Society Open Science (2024)

[231] A. Uhrmacher et al., *Context, Composition, Automation, And Communication–the C2ac Roadmap for Modeling and Simulation*, Arxiv Preprint Arxiv:2310.05649 (2023)

[232] D. Moreau, K. Wiebels, and C. Boettiger, *Containers for Computational Reproducibility*, Nature Reviews Methods Primers **3**, 50 (2023)

[233] S. F. J. Apostal, D. Apostal, and R. Marsh, *Containers and Reproducibility in Scientific Research*, in *2018 IEEE International Conference on Electro/information Technology (EIT)* (2018), pp. 525–530

[234] R. Ewald and A. M. Uhrmacher, *SESSL: A Domain-Specific Language for Simulation Experiments*, ACM Transactions on Modeling and Computer Simulation (TOMACS) **24**, 1 (2014)

[235] P. A. Fishwick, *Web-Based Simulation: Some Personal Observations*, in *Proceedings of the 28th Conference on Winter Simulation* (IEEE Computer Society, Coronado, California, USA, 1996), pp. 772–779

[236] J. Byrne, C. Heavey, and P. Byrne, *A Review of Web-Based Simulation and Supporting Tools*, Simulation Modelling Practice and Theory **18**, 253 (2010)

[237] A. Rossberg, WebAssembly Core Specification, 2019

[238] X. Klemenschits, P. Manstetten, L. Filipovic, and S. Selberherr, *Process Simulation in the Browser: Porting Viennats Using Webassembly*, in *2019 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, Vol. 0 (2019), pp. 1–4

[239] R. Fujimoto, *Parallel and Distributed Simulation*, in *2015 Winter Simulation Conference* (2015), pp. 45–59

[240] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software (TOMS) **5**, 308 (1979)

[241] L. S. Blackford et al., *An Updated Set of Basic Linear Algebra Subprograms (BLAS)*, ACM Transactions on Mathematical Software **28**, 135 (2002)

[242] B. Kågström, P. Ling, and C. Van Loan, *GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark*, ACM Transactions on Mathematical Software (TOMS) **24**, 268 (1998)

[243] S. Kestur, J. D. Davis, and O. Williams, *Blas Comparison on Fpga, Cpu and Gpu*, in *2010 IEEE Computer Society Annual Symposium on VLSI* (2010), pp. 288–293

[244] R. M. Fujimoto, Performance Measurements of Distributed Simulation Strategies., 1987

[245] V. Bonnet, Benchmarking Parallel Discrete Event Simulations, 2017

[246] EunJung Park, S. Eidenbenz, N. Santhi, G. Chapuis, and B. Settlemyer, *Parameterized Benchmarking of Parallel Discrete Event Simulation Systems: Communication, Computation, And Memory*, in *2015 Winter Simulation Conference*, Vol. 0 (2015), pp. 2836–2847

[247] E. Glinsky and G. Wainer, *Devstone: A Benchmarking Technique for Studying Performance of DEVS Modeling and Simulation Environments*, in *Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications* (2005), pp. 265–272

[248] P. A. Wilsey, *Some Properties of Events Executed in Discrete-Event Simulation Models*, in *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2016), pp. 165–176

[249] P. Crawford, S. J. Eidenbenz, P. D. Barnes, and P. A. Wilsey, *Some Properties of Communication Behaviors in Discrete-Event Simulation Models*, in *2017 Winter Simulation Conference* (2017), pp. 1025–1036

[250] S. Kane, S. Gupta, and P. A. Wilsey, *Analyzing Simulation Model Profile Data to Assist Synthetic Model Generation*, in *2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (2019), pp. 1–10

[251] J. Nutaro and O. Ozmen, *Race Conditions and Data Partitioning: Risks Posed by Common Errors to Reproducible Parallel Simulations*, SIMULATION **99**, 417 (2023)

[252] G. Marsaglia and others, *Xorshift Rngs*, Journal of Statistical Software **8**, 1 (2003)

[253] A. Piccione, P. Andelfinger, and A. Pellegrini, *Hybrid Speculative Synchronisation for Parallel Discrete Event Simulation*, in *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2023), pp. 84–95

[254] R. Marotta, A. Pellegrini, and P. Andelfinger, *Follow the Leader: Alternating CPU/GPU Computations in PDES*, (2024)

[255] M. Jeschke and R. Ewald, *Large-Scale Design Space Exploration of Ssa*, in *Computational Methods in Systems Biology: 6th International Conference CMSB 2008, Rostock, Germany, October 12-15, 2008. Proceedings 6* (2008), pp. 211–230

[256] P. Städter, Y. Schälte, L. Schmiester, J. Hasenauer, and P. L. Stapor, *Benchmarking of Numerical Integration Methods for ODE Models of Biological Systems*, Scientific Reports **11**, 2696 (2021)

[257] B. Shaikh et al., *Biosimulators: A Central Registry of Simulation Engines and Services for Recommending Specific Tools*, Nucleic Acids Research **50**, (2022)

[258] L. Herrmann, Comparing the Performance of Stochastic Simulators Using a Synthetic Benchmark, 2023

[259] K. Choi, J. K. Medley, M. König, K. Stocking, L. Smith, S. Gu, and H. M. Sauro, *Tellurium: An Extensible Python-Based Modeling Environment for Systems and Synthetic Biology*, Biosystems **171**, 74 (2018)

[260] J. H. Abel, B. Drawert, A. Hellander, and L. R. Petzold, *GillesPy: A Python Package for Stochastic Model Building and Simulation*, IEEE Life Sciences Letters **2**, 35 (2016)

[261] V. Andreani, *Rebop "Https://github.com/armavica/rebop"*, (2023)

[262] O. Ostrenko, P. Incardona, R. Ramaswamy, L. Brusch, and I. F. Sbalzarini, *pSSAlib: The Partial-Propensity Stochastic Chemical Network Simulator*, PLOS Computational Biology **13**, 1005865 (2017)

[263] K. R. Sanft, S. Wu, M. Roh, J. Fu, R. K. Lim, and L. R. Petzold, *StochKit2: Software for Discrete Stochastic Simulation of Biochemical Systems with Events*, Bioinformatics **27**, 2457 (2011)

[264] C. Li et al., *BioModels Database: An Enhanced, Curated and Annotated Resource for Published Quantitative Kinetic Models*, BMC Systems Biology **4**, (2010)

[265] B. G. Olivier and J. L. Snoep, *Web-Based Kinetic Modelling Using JWS Online*, Bioinformatics **20**, 2143 (2004)

[266] R. Ewald and A. M. Uhrmacher, *Automating the Runtime Performance Evaluation of Simulation Algorithms*, in *Proceedings of the 2009 Winter Simulation Conference (WSC)* (2009), pp. 1079–1091

[267] S. G. Riva, P. Cazzaniga, M. S. Nobile, S. Spolaor, L. Rundo, D. Besozzi, and A. Tangherloni, *Smgen: A Generator of Synthetic Models of Biochemical Reaction Networks*, Symmetry **14**, 119 (2022)

[268] M. A. Kochen, H. S. Wiley, S. Feng, and H. M. Sauro, *Sbbadger: Biochemical Reaction Networks with Definable Degree Distributions*, Bioinformatics **38**, 5064 (2022)

[269] T. Köster, T. Warnke, and A. M. Uhrmacher, *Generating Fast Specialized Simulators for Stochastic Reaction Networks via Partial Evaluation*, ACM Transactions on Modeling and Computer Simulation (TOMACS) **32**, 1 (2022)

[270] T. Köster, P. Henning, and A. M. Uhrmacher, *Potential Based, Spatial Simulation of Dynamically Nested Particles.*, BMC Bioinformatics **20**, (2019)

[271] T. Köster, P. J. Giabbanelli, and A. M. Uhrmacher, *Performance and Soundness of Simulation: A Case Study Based on a Cellular Automaton for in-Body Spread of HIV*, in *Winter Simulation Conference (WSC 2020)* (IEEE Computer Society, 2020), pp. 2281–2292

[272] T. Köster, L. Herrmann, P. Andelfinger, and A. M. Uhrmacher, *GPU-Accelerated Simulation Ensembles of Stochastic Reaction Networks*, in *Winter Simulation Conference (WSC 2022)* (IEEE, 2022), pp. 2570–2581

[273] T. Köster, P. Henning, T. Warnke, and A. Uhrmacher, *Expressive Rule-Based Modeling and Fast Simulation for Dynamic Compartments*, Plos One **19**, 312813 (2024)

[274] T. Köster, O. Reinhardt, M. Hinsch, J. Bijak, and A. M. Uhrmacher, *A Fast Embedded Language for Continuous-Time Agent-Based Simulation*, Journal of Artificial Societies and Social Simulation **27**, 10 (2024)

[A] Generating Fast Specialized Simulators for Stochastic Reaction Networks

Domain-specific modeling languages allow a clear separation between simulation model and simulator and, thus, facilitate the development of simulation models and add to the credibility of simulation results. Partial evaluation provides an effective means for efficiently executing models defined in such languages. However, it also implies some challenges of its own. We illustrate this and solutions based on a simple domain-specific language for biochemical reaction networks as well as on the network representation of the established BioNetGen language. We implement different approaches adopting the same simulation algorithms: one generic simulator that parses models at runtime and one generator that produces a simulator specialized to a given model based on partial evaluation and code generation. For the purpose of better understanding, we additionally generate intermediate variants, where only some parts are partially evaluated. Akin to profile-guided optimization, we use dynamic execution of the model to further optimize the simulators. The performance of the approaches is carefully benchmarked using representative models of small to large biochemical reaction networks. The generic simulator achieves a performance similar to state-of-the-art simulators in the domain, whereas the specialized simulator outperforms established simulation tools with a speedup of more than an order of magnitude. Technical limitations in regard to the size of the generated code are discussed and overcome using a combination of link-time optimization and code separation. A detailed performance study is undertaken, investigating how and where partial evaluation has the largest effect.

[269] T. Köster, T. Warnke, and A. M. Uhrmacher, *Generating Fast Specialized Simulators for Stochastic Reaction Networks via Partial Evaluation*, ACM Transactions on Modeling and Computer Simulation (TOMACS) **32**, 1 (2022)

Reaction Networks via Partial Evaluation

TILL KÖSTER, TOM WARNKE, and ADELINDE M. UHRMACHER, University of Rostock

Domain-specific modeling languages allow a clear separation between simulation model and simulator and, thus, facilitate the development of simulation models and add to the credibility of simulation results. Partial evaluation provides an effective means for efficiently executing models defined in such languages. However, it also implies some challenges of its own. We illustrate this and solutions based on a simple domain-specific language for biochemical reaction networks as well as on the network representation of the established BioNetGen language. We implement different approaches adopting the same simulation algorithms: one generic simulator that parses models at runtime and one generator that produces a simulator specialized to a given model based on partial evaluation and code generation. For the purpose of better understanding, we additionally generate intermediate variants, where only some parts are partially evaluated. Akin to profile-guided optimization, we use dynamic execution of the model to further optimize the simulators. The performance of the approaches is carefully benchmarked using representative models of small to large biochemical reaction networks. The generic simulator achieves a performance similar to state-of-the-art simulators in the domain, whereas the specialized simulator outperforms established simulation tools with a speedup of more than an order of magnitude. Technical limitations in regard to the size of the generated code are discussed and overcome using a combination of link-time optimization and code separation. A detailed performance study is undertaken, investigating how and where partial evaluation has the largest effect.

CCS Concepts: • **General and reference** → **Performance**; • **Software and its engineering** → Automated static analysis; Dynamic analysis; **Source code generation**; *Domain specific languages*; • **Mathematics of computing** → Markov-chain Monte Carlo methods; • **Computing methodologies** → *Modeling methodologies;*

Additional Key Words and Phrases: Simulation, modelling, high performance, code generation, partial evaluation, SSA

ACM Reference format:
Till Köster, Tom Warnke, and Adelinde M. Uhrmacher. 2022. Generating Fast Specialized Simulators for Stochastic Reaction Networks via Partial Evaluation. *ACM Trans. Model. Comput. Simul.* 32, 2, Article 10 (February 2022), 25 pages.
https://doi.org/10.1145/3485465

Financial support was provided by the Deutsche Forschungsgemeinschaft (DFG) research grant ESCeMMo (UH-66/13).
Authors' address: T. Köster, T. Warnke, and A. M. Uhrmacher, University of Rostock, Albert-Einstein-Str. 22, Rostock, Germany, 18059; emails: {till.koester, tom.warnke, adelinde.uhrmacher}@uni-rostock.de.

1049-3301/2022/02-ART10 $15.00
https://doi.org/10.1145/3485465

1 INTRODUCTION & BACKGROUND

To simulate a given model, two general approaches can be distinguished. First, the model can be directly implemented by intertwining the model description with the machinery to execute it. In such a *specialized* approach, the implementation can be tailored to the model by hand, exploit domain- or model-specific optimizations, and, consequently, deliver computationally efficient simulation. However, specialized implementations of algorithms are hard to test, maintain, or reuse [23]. The second approach utilizes a separation of model (an abstract surrogate of a system) and simulator (an algorithm that, given a model as input, produces output) [44]. The simulation algorithms implemented in this way are *generic* referring to a specific class of models. Whereas generic algorithms are easier to test, maintain, and reuse than specialized algorithms [21], they can exploit less model-specific optimizations and are, therefore, less computationally efficient.

Generic simulators have been successfully applied to *modeling languages* or *modeling formalisms* such as DEVS [43], Petri nets [18], or rule-based languages [10]. In the field of cell biology, standardized model exchange languages such as the **Systems Biology Markup Language (SBML)** [24] and CellML [32] facilitate the development of generic simulators [16]. Modeling languages define syntax and semantics, and it is not uncommon to provide formal definitions [20]. A generic simulation algorithm can rely on the syntax to parse a model description and then execute a simulation run according to the semantics. This approach corresponds to implementing an *interpreter* for a general-purpose programming language. Consequently, it incurs some interpretative overhead when compared to an implementation specialized to the model. A specialized implementation is comparable to the program produced by a *compiler*, where the model description plays the part of the input source code.

Implementing a generic simulator in the interpretative style is straightforward. In contrast, the job of the compiler (producing a specialized simulation algorithm) is typically done manually. The issue of automating this job is tackled by the fields of *partial evaluation* or *program specialization* [12, 22]. The idea of partial evaluation is to automatically transform an algorithm by partially applying it to some of its inputs. Whereas partial evaluation is a well-researched topic, so far, it has found limited use in modeling and simulation.

In this paper, we present an approach to high-performance simulation of biochemical reaction networks expressed in a rule-based language by applying partial evaluation. This application domain has several beneficial properties for our approach. First, there is a large collection of existing models that can be executed with newly developed algorithms. Second, the core syntax and semantics of rule-based languages can be kept quite simple (although more complex languages exist as well), which allows us to focus on the methodological novelties and less on the languages. Third, executing simulation runs of rule-based languages is well-researched, and existing performance studies [16] facilitate evaluating our approach.

Specifically, our contributions are the following:

- We implement a variant of Gillespie's **stochastic simulation algorithm (SSA)** [15] in two ways. First, as an interpretative generic algorithm that parses a model at runtime. Second, as a compiler-like code generator that produces code for a specialized model-specific algorithm.
- We tune the parameters of the simulation algorithms via profile-guided optimization by inferring the optimal algorithmic configuration from an instrumented model execution.
- We integrate both implementations with the established BioNetGen simulation package. Thus, our implementations are applicable to a wide variety of models.
- We conduct a detailed performance analysis based on established benchmark models from the literature. Our results show that the benefits of code generation quickly outweigh the additional generation and compilation effort when compared to a generic algorithm.

This paper is an extended version of the paper published at the PADS conference [27]. The initial work only generated a single source code file. Therefore, compilation times were exceptionally high, limiting the size of the investigated model. Additionally, the performance analysis of the partial evaluation was far less in-depth than it is here. Furthermore, the algorithmic capabilities were extended, making use of the code generation technology, for example using arbitrary complex guard expressions at no additional cost. We have also investigated more variants of the code generation, including different algorithms as well as different degrees of partial evaluation. Finally, in this paper, we also evaluate the impact of different compilation strategies, like link-time optimization.

Some of the results shown in this paper were independently verified using our publicly available code as part of the PADS reproducibility initiative [6].

2 BACKGROUND

The main idea of this paper is applying partial evaluation to the simulation of rule-based modeling languages. We start by shortly introducing the core ideas of both fields. Partial evaluation stems from the area of programming language theory, whereas rule-based modeling is employed in computational biology.

2.1 Partial Evaluation

The idea of partial evaluation dates back at least to Futamura's work in the early 1970s [12]. Futamura describes different levels of program specialization, starting with some generic computation and gradually becoming more specialized. The key idea is to provide a computation α to automatically increase the level of specialization by partially evaluating a computation for some of its inputs. Concretely, a computation π with $m + n$ inputs

$$\pi(i_1, \ldots, i_m, i_{m+1}, \ldots, i_{m+n})$$

can be specialized by supplying values for the first m inputs to α, resulting in a new computation π' that still takes the remaining n inputs.

$$\pi(i_1, \ldots, i_m, i_{m+1}, \ldots, i_{m+n}) = \alpha(\pi, i_1, \ldots, i_m)(i_{m+1}, \ldots, i_{m+n}) = \pi'(i_{m+1}, \ldots, i_{m+n}).$$

The equalities in this equation express the correctness of the transformation [22]. Note that π' depends on the concrete choice for $i_1, \ldots, i_m$, and the second equality only holds for these $i_1, \ldots, i_m$, although our notation omits this. The values for $i_1, \ldots, i_m$ are "hardcoded" into π' by α. These fixed inputs are called *static* inputs, whereas the residual inputs of π' are called *dynamic* inputs. As a consequence, segments of π that only depend on $i_1, \ldots, i_m$ can already be evaluated by α when transforming π to π'.

A short, abstract example is the following recursive computation to compute x^n:

```
pow(x, n) = if (n = 0) then 1 else x * pow(x, n - 1)
```

For any concrete value of `n`, for example 3, this computation can be transformed to a new computation that computes x^3:

```
pow3(x) = x * x * x
```

In this case, the recursion could be evaluated by α, leaving a function with only one input that executes only two multiplications. As exemplified by this transformation, the transformed computation is typically simpler (e.g., avoids recursion) and more computationally efficient (e.g., by requiring fewer calculations). However, α itself is a computation with some complexity and runtime.

We can now return to the concepts of interpreters and compilers and express the relation between them with specialization steps. First, an interpreter *int* is a computation that takes as its input the program source code p and p's inputs $i_1, \ldots, i_n$. When α specializes *int* with respect to p, a computation taking only $i_1, \ldots, i_n$ remains—the executable program. Another layer of specialization then results in a computation that (for a static interpreter) takes the program code and produces the executable—a compiler.

$$\begin{aligned} & int(p, i_1, \ldots, i_n) && \text{Interpreter} \\ = \; & \alpha(int, p)(i_1, \ldots, i_n) && \text{Executable} \\ = \; & \alpha(\alpha, int)(p)(i_1, \ldots, i_n) && \text{Compiler} \end{aligned}$$

The compiler usually provides more efficient execution than interpreters, for example by type-checking the program during compilation and then producing an executable without runtime type-checking. However, interpreters are often simpler as they do not need to distinguish between compile-time and runtime [22].

We can apply the same idea to the notions of generic and specialized simulators. A generic simulator *sim* is a computation that takes a model m, model parameters $p_1, \ldots, p_n$, and a random seed r as input and computes some sort of simulation output. By considering the model m as a static input and the remaining parameters as dynamic, we can obtain a specialized simulator via α. The computation that generates a specialized simulator for a given model (and a static generic simulator) is obtained with one additional level of specialization.

$$\begin{aligned} & sim(m, p_1, \ldots, p_n, r) && \text{Generic Simulator} \\ = \; & \alpha(sim, m)(p_1, \ldots, p_n, r) && \text{Specialized Simulator} \\ = \; & \underline{\alpha(\alpha, sim)}(m)(p_1, \ldots, p_n, r) && \text{Simulator Generator} \end{aligned}$$

Implementing a *simulator generator* [25] is the main focus of this paper. Although the specialization α is useful to define the simulator generator, we implement the underlined part above directly rather than α. This simulator generator itself is a computation with a certain runtime or computational cost, so the specialization involves a trade-off. However, when a simulation model is used in multiple simulation runs, the one-time cost of generating a specialized simulator should become negligible.

Similar to how a compiler works, a simulator generator is able to introduce model-specific optimizations into the generated simulator. In particular, it can exploit the properties of the model to speed up the simulation. Typically, these properties can be determined by syntactic or semantic analysis of the model. For example, a generic algorithm must be able to run models with arbitrarily large states and, therefore, use a flexible data structure. A specialized algorithm, on the other hand, can hardcode the state representation and avoid indirections when accessing it.

In practice, partial evaluation has been applied to speed up computations, often by simplifying them. For example, Schultz et al. [39] presents an approach that employs partial evaluation to remove object-orientation from Java code to increase computational performance. In a similar fashion, Zeng [45] applies partial evaluation to remove concurrency or recursion from programs written in **domain-specific languages (DSLs)**. Other works focus on developing methods for effectively integrating partial evaluation into programs. Rompf and Odersky [37] present a library that exploits Scala's expressive type system to distinguish expressions evaluated during program generation and during program execution. Leissa et al. [28] presents a DSL that allows partial evaluation targeting CPUs as well as GPUs and apply it to computation-intensive problems like image processing or genome sequencing.

In our approach, we do not implement the specialization α itself, but its specialization to the stochastic simulation of biochemical reaction networks. This corresponds to the underlined part above, with m being a (rule-based) model of a reaction network and *sim* being a stochastic simulation algorithm.

2.2 Rule-based Modeling, Reaction Networks, and Stochastic Simulation Algorithms

For modeling biochemical reaction networks, an explicit model representation that is clearly separated from the simulation algorithm is widely adopted. Among the variety of available languages and formalisms for modeling biochemical reaction networks, *reaction-based* and *rule-based* modeling has become popular.

In reaction-based modeling, a system is described by enumerating chemical species and reactions between them. A reaction consists of the reacting species (the reactants), the resulting species (the products), and an expression for the reaction rate, which is usually a function of the reactant population sizes. Rule-based modeling extends this by equipping species with attributes and defining reaction rules instead of reactions [10]. Reaction rules specify the reactants as patterns, and the products and rate expression depend on the matched reactants.

If the domain of the attribute values is finite, a rule-based model can be transformed into an equivalent reaction-based model. For example, consider a species A with three attributes, each of which can take one of the two values p and u. The reaction rule

$$A(u, x_1, x_2) \longrightarrow A(p, x_1, x_2) \mathbin{@} k_1 \cdot \#A(u, x_1, x_2)$$

uses the variables x_1 and x_2. Substituting the possible values for both variables yields 4 reactions involving 8 species:

$$\begin{aligned} Auuu &\longrightarrow Apuu \mathbin{@} k_1 \cdot \#Auuu \\ Auup &\longrightarrow Apup \mathbin{@} k_1 \cdot \#Auup \\ Aupu &\longrightarrow Appu \mathbin{@} k_1 \cdot \#Aupu \\ Aupp &\longrightarrow Appp \mathbin{@} k_1 \cdot \#Aupp \end{aligned}$$

Although a high number of attributes can lead to a combinatorial explosion of the number of species and reactions, many rule-based models can be transformed into reaction networks this way. Such a reaction network then defines a population-based CTMC [1, 10, 20]. Consequently, the standard approach for simulating biochemical reaction networks is sampling this CTMC with a **Stochastic Simulation Algorithm (SSA)**.

Gillespie presented an SSA for reaction networks, the Direct Method, in 1977 [15]. Since then, many variations of this algorithm have been developed, including the Next Reaction Method [13], the **Optimized Direct Method (ODM)** [4], as well as the Sorting Direct Method [30]. Here, the use of smarter data structures increased performance. Recently, also the particular nature of mass action kinetics is exploited to gain performance, e.g., in the rejection-based sampling algorithm [42]. Gupta and Mendes [16] authored a comprehensive review that tests the performance of state-of-the-art implementations. In this review, BioNetGen [17] and pSSAlib [34] are shown to be the fastest. Our approach combines insights from this research with the idea of partial evaluation.

3 RELATED WORK

A few approaches that use code generation or compile-time optimization to speed up the simulation of reaction networks have been proposed.

Meyer et al. [31] augment the Java ML-Rules simulation algorithm with runtime code generation. The original simulation algorithm stored rate expressions as abstract syntax trees and evaluated

them in an interpretative fashion. To speed up the evaluation of rate expressions, each abstract syntax tree is replaced with generated Java code representing the same expression. Depending on the model, this change reduced the overall simulation time by 5% to 42%. In contrast to our work, this approach is less intrusive—due to the reflective capabilities of the JVM, the code generation can be integrated transparently into an existing simulation algorithm. On the other hand, it is also less effective—as it only transforms rate expressions, it is not able to exploit model-specific memory management, for example.

The **stochastic simulation compiler SSC** [29] focuses on "*combining a higher level specification required for modeling larger systems with the ability to model spatially heterogeneous systems*". To counter the increased computational complexity caused by spatial heterogeneity, SSC employs code generation to produce efficient model-specific simulators. Similar to our approach, the input model can be defined in a custom DSL or be automatically translated from BioNetGen. Whereas we generate C++ code that is then compiled, SSC directly generates native x86 assembly and links it with a C runtime. An old (2009) compiled binary for SCC is available. However, the source code is not.

LibRoadRunner [41] simulates models defined in SBML, which is a general description format for biochemical reaction networks. In contrast to our approach, libRoadRunner relies on LLVM, a compiler framework, and integrates its code generation with LLVM's **Just-In-Time (JIT)** compiler. This yields native machine code for model-specific SBML simulators. However, for the specific subset of simulation models that we consider in this work, namely static stochastic models, libRoadRunner was shown to not be competitive [16].

The simulation of reaction networks is in part related to that of Petri nets. For these models, code generation has been explored [35], however mostly not for performance improvements, but for usages on specific hardware like microcontrollers [3] or integration with a 3D game engine [5].

The work that is most closely related to ours is a paper from 2008 by Keller et al. [25]. They also propose the generation of specialized simulators based on partial evaluation of a generic algorithm and achieve performance gains through this strategy. We expand on this work in several directions. First, we integrate our implementation to the existing state-of-the-art tool BioNetGen, demonstrating the practical relevance of the approach. Second, we provide an in-depth performance analysis of the effects of partial evaluation and different compilation strategies. Third, we illuminate the interaction of partial evaluation with algorithmic improvements of the SSA, including profile-guided optimization of model-specific algorithm configuration. Fourth, we relate the performance gains we achieve to the top state-of-the-art algorithms, proving that a simulator powered by partial evaluation is able to surpass all of them.

Code generation of several model components and their linking has also been a part of the Möbius-framework for complex systems modelling [8]. There, the different components of the model are compiled independently and linked thereafter, similar to what we will be doing to improve our compilation time.

Automatic model transformation has also been used more generally in model development for simulations, for example in model-driven development [7].

4 OVERVIEW OF OUR APPROACH

The technical framework for our approach is shown in Figure 1. It contains several input languages and several execution options. The common layer between inputs and execution is a machine-readable representation of the complete reaction network specified in a TOML file. TOML is a frequently used simple textual data markup format. **TOML** stands for ***Tom's Obvious, Minimal Language***. Note that, in addition to the steps shown in Figure 1, it is also possible to process the model representations further, for example with methods for model reduction or abstraction [36].

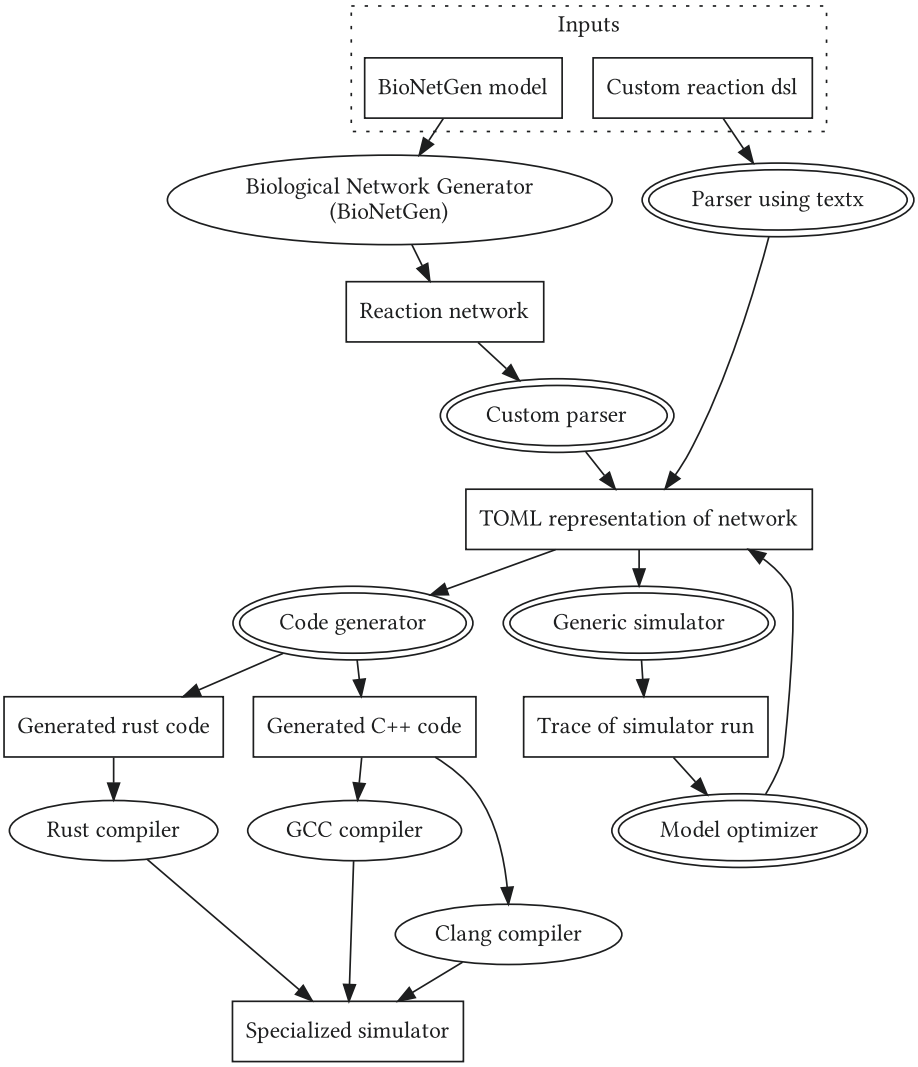


Fig. 1. An overview of the technical framework and involved tools as well as their interdependence. Software tools are denoted by ellipses, own implementations are highlighted. Rectangles are for files and (generated) artefacts.

4.1 Input Languages

Currently, we accept two kinds of input models. A custom DSL presents one option to create TOML representations. This DSL does not support attributed species or rule-based modeling but enables writing reaction networks in a human-readable syntax. A model specified in the DSL consists of an enumeration of species, the initial population of each species, and a set of reactions. As already introduced in Section 2.2, reactions look like

$$\underbrace{A + B}_{\text{Left side}} \longrightarrow \underbrace{C}_{\text{right side}} \quad @ \underbrace{\#A \cdot \#B \cdot \overbrace{3.8}^{\text{Mass action rate}}}_{\text{reaction propensity}}$$

Each reaction specifies the left side species that turn into the right side species at a rate which is given as a function of the amount of the reactant population and the reaction rate constant,

ALGORITHM 1: Direct Method
t: current simulation time,
t_{end}: the end time of the simulation run
$X = (x_1, x_2, \ldots, x_n) \in \mathbb{N}^n$: the state vector.
$R = \{R_1, R_2, \ldots, R_m\}$: the set of reactions.
$\mathbf{v_i} \in \mathbb{Z}^n$: state change vector of reaction R_i.
$a_1(X) \ldots a_m(X)$: propensities of the reactions $R_1, \ldots R_m$.
$a_0(X)$: sum of all reaction propensities.

```
1  while t < t_end do:
2    // Calculate propensity sum of all reactions
3    a_0(X) := ∑_{i=1}^{m} a_i(X)
4
5    // Select a reaction to be executed
6    x := U(0, a_0(X))
7    i := min{j|j ∈ {1...m} ∧ ∑_{i=1}^{j} a_i(X) > x}
8
9    // Advance simulation time by sampling from an
10   // exponential distribution with rate parameter λ = a_0(X)
11   t := t + Exp(a_0(X))
12
13   // Execute selected reaction
14   X := X + v_i
```

yielding the so-called propensity of the reaction. We have implemented a parser for this DSL using the Python textx library[1] to produce the TOML file.

In addition, we support the execution of models specified in the **BioNetGen language (BNGL)** [10]. BioNetGen is a popular open-source software for the specification and simulation of rule-based models of complex biochemical systems, including signal transduction, metabolic, and genetic regulatory networks. BNGL models can be exported into different formats, such as SBML or the MATLAB language [17]. One of the core functionalities of BioNetGen is the generation of the underlying reaction network as described in Section 2.2. The reaction network serves as input for the built-in population-based network simulators of BioNetGen. Our framework includes a parser that transforms BioNetGen's reaction network format into our TOML file format. Thus, our framework uses the reaction network as expanded by BioNetGen as input to generate specialized simulators.

4.2 Simulation with the (Optimized) Direct Method

The network description in TOML serves as input for the generic simulator as well as for the code generator that generates the specialized simulators. All our simulators implement variants of the Direct Method, the stochastic simulation algorithm introduced by Gillespie [15] (Algorithm 1).

A number of algorithmic optimizations of the Direct Method have been proposed. In this paper, we focus on two of those optimizations: maintaining a dependency graph of reactions and optimally ordering reactions.

[1] textX is available at https://textx.github.io/textX/ and is inspired by the popular Xtext framework (https://www.eclipse.org/Xtext/).

- Using a dependency graph [13] avoids calculating all reaction propensities at every time step (line 3 of Algorithm 1). Instead, the propensity of all reactions as well as their sum is stored between steps. Whenever a species amount is changed, the propensity is only recalculated for the reactions that are dependent on this species, and the global propensity sum is changed accordingly.
- The ordering of reactions can be optimized by sorting reactions such that reactions with a high propensity occur first. Since fewer reactions need to be traversed to reach the threshold (line 7 in Algorithm 1), the selection of a reaction is significantly sped up.

The combination of these optimizations has been shown to improve performance significantly in the **Optimized Direct Method (ODM)** [4]. We always use the dependency graph for all simulators.

A central step in the Direct Method is the generation of random numbers. An exponential random number is needed for the timestep (line 6 in Algorithm 1) and a uniform random number for the reaction selection (line 11 in Algorithm 1). All Rust implementations use the PCG random number generator from the Rust Rand Library [33]. By default, Rust uses advanced random number generators designed for applications in cryptography. For our purposes, a PCG is sufficient, as it already provides high-quality numbers but is significantly faster. For the C++ implementation, we have used both Mersenne Twister, which is a recommended C++ random number generator and PCG. Mersenne Twister is slightly slower than PCG, but that makes little difference for most models.

4.2.1 Generic Simulators. A generic simulator was implemented in both Rust and C++. It reads the TOML file containing the model description and executes it according to Algorithm 1. The structure of the model is stored in heap-allocated arrays that are created at runtime after parsing the TOML model description. The entire simulator has about 260 lines of code, including all parsing and output.

4.2.2 Generated Specialized Simulators. We create the specialized simulators by explicit code generation. This is done by a Python3 program, the Code Generator (see Figure 1), using jinja2[2] templates for code generation. All reactions are hardcoded into the specialized simulators. Both C++ or Rust code can be generated.

Rust was chosen as it creates fast native code (through its compilation backend LLVM) with strong static guarantees. In addition, the Rust ecosystem includes a good compilation and dependency management ecosystem (crates.io and Cargo), which enables compilation of the generated code on various hard- and software environments with minimal effort. In the generated Rust code, a simulator is an object that only contains the propensities (the calculated rates) of the reactions as well as the population sizes of the species. Both of them are stored as stack arrays. In fact, the entire simulator never performs any heap allocation in any performance-critical part of the simulation. In the main step function the index of the next reaction is calculated (line 7 of Algorithm 1). Whereas the generic simulator uses this index to look up the reaction to execute in a dynamic data structure, the generated code immediately jumps to the execution of the reaction at the determined index. Additionally, it is hardcoded which reactions are affected, and their respective propensities are also updated (if using a dependency graph is enabled). The recalculation of a reaction's propensity is a small function that can be inlined by the compiler if needed. There is never the need to look up any dependency graph, and the compiler is free to reorder any calculations as it sees fit.

[2] palletsprojects.com/p/jinja/.

C++ is a well-established language known for creating fast assembly. One advantage of C++ is its long development in compiler optimization and infrastructure. We will exploit some features when we move towards larger models later in this paper. Due to compile overhead, these larger models were not tractable using compilation with Rust. Therefore, more advanced methods of compilation, as well as some of the advanced model features, are only supported in the C++ code generator. The generated C++ simulator is similar to the Rust one, but all functions and variables live at global scope. Other than the random number generation, which uses their respective standard libraries, the algorithmic implementation is identical. There are no objects used for the C++ code (except the random number calculator).

5 OPTIMIZING THE SIMULATION

In this section, we will discuss the different kinds of optimizations made. First, we will discuss optimizations from the area of partial evaluation. These methods invest additional effort when generating the simulator to improve the performance of the generated simulator. These optimizations lay on a spectrum—there is a trade-off between increased up-front effort and improved simulator performance. Furthermore, we present some additional algorithmic optimizations that we have investigated in Section 5.4.

5.1 Profile-guided Optimization for Reaction Ordering

Profile-guided optimization [40] is a type of optimization for repeatedly executed and performance-critical programs. For this, an instrumented variant of a program is executed once while recording various statistics on the execution. This record of typical execution behavior is then used to further optimize the code. A typical example would be branch prediction, where more common choices can be listed first.

We employ the idea of profile-guided optimization to integrate the optimal ordering of reactions as proposed in the Optimized Direct Method (Section 4.2) into the generation of specialized simulators. In particular, we infer an optimal reaction ordering based on profile data from a pilot run. To do this, we use a simple heuristic of counting how often a reaction fired during a sample execution of the model (the pilot run). Then we regenerate the entire model (i.e., the TOML file), but sort the reactions by how often they fired. This basic strategy is already helpful in reducing the reaction selection times. In principle, of course, reaction propensities can differ between simulation runs. However, it is observed in the literature [4], as well as in our experience, that one pilot run tends to lead to a good average behavior for simulation execution. If changes were more dynamic, the Sorting Direct Method [30], in which the reaction order is changed at runtime to create more optimal reaction selection, would provide an alternative. This is the algorithm that was chosen by BioNetGen.

It makes sense to execute the pilot run with the generic simulator, as we only need the results from one run. It is typically slower than the specialized simulators but requires no upfront compilation.

5.2 Compiler Optimizations

As shown in our initial investigation [27], one of the challenges of generating simulators is the size of the generated code. Generating and compiling a single source code file allows for maximum optimizations during compilation but limits the size of the input models. Large input models can lead to compilers raising internal errors and also to compile times and memory needs that are not practical for real-world usage. In addition, it complicates parallelizing the compilation.

To mitigate this, we can generate multiple source code files. This includes separate files for initialization, IO, as well as the reactions of the system. Now for each reaction, different functions

serve as an interface to other reactions, like reaction execution or rate calculation. However, it is also not practical to generate one file for each reaction, as too many files also lead to technical issues with compilation and linking, such as the limit in the length of a bash gcc call. Therefore we have made the following choice: We create at most 2,000 files for the different reactions. If there are more than 2,000 reactions, we put multiple reactions into a single file, constituting one unit of compilation. This way, we can avoid having too few as well as too many source code files.

The generation of many separate source code files allows parallel compilation and also reduces memory needs. On modern machines, this leads to large compilation speedups, as it is essentially an embarrassingly parallel problem. However, these advantages in compile performance come at a disadvantage for the runtime performance. These separate units of compilation are later only linked together for the final executable. Some common and important compilation optimizations like inlining are limited to units of compilation and cannot be applied during linking. Therefore the final execution speed suffers.

To combine the optimization potential of a single source code file with the speed of generating and compiling many source code files, we exploit **link-time optimization (LTO)**. Here the units of compilation are only partially compiled into an intermediate representation, which can be done in parallel. Subsequently, the intermediate representations are linked, and the final executable is produced. During this step, which is not parallel, LTO improves the performance of the executable at the cost of more linking time. We have included results for all three modes of generation and compilation in the results section.

5.3 Specifying Parameters at Runtime

Partial evaluation does not necessarily include all model parameters. For maximum performance of a simulation run with a specific parametrization, all model parameters should be hardcoded into the generated simulator. However, in some situations, for example when running a parameter sweep with a model, some model parameters may change frequently. To avoid having to recompile the complete model for every parameter change, model parameters can be set dynamically. Instead of hardcoding them into the simulator, these parameters have to be specified at runtime. Again, this means a performance trade-off, as with more parameters known at compile-time, more expressions can be simplified, leading to faster execution. We have analyzed this trade-off in the results section.

5.4 Algorithmic Optimizations of the Specialized and Generic Simulator

To compare the specialized and the generic simulator, we use the same algorithm with very similar implementation in both simulators. To be competitive with other tools in the domain and to make our tool useful for the community, we have included some algorithmic optimizations compared to a naïve realization in both implementations.

5.4.1 Redundant Operation & Reaction Removal. When writing reactions frequently, some entities remain on both sides and/or are unchanged. Our parser will, whenever this is the case, remove the entity from both sides, i.e. we take

$$A + B \longrightarrow B \,@\, 7 \cdot \#A \cdot \#B$$

and transform it into

$$A \longrightarrow \quad @\, 7 \cdot \#A \cdot \#B.$$

This is always done for both simulators. We did not measure the performance difference of this minor optimization.

We also statically analyze all reactions in the system. It is common practice in models to have some rates or constants set to zero by the modeler to disable or enable features within the model. These reactions are skipped when parsing.

5.4.2 Tree-based Reaction Management. To compare different approaches of reaction selection, we have also implemented a variant of the (optimized) Direct Method, where the propensities are stored in a tree instead of a vector. Here reaction selection has logarithmic instead of linear complexity, but on the other hand, reaction rate updates also have logarithmic complexity instead of constant time. This structure is effectively a Fenwick tree [11, 38] and has been used for SSA simulation in the past [9]. However, its application in conjunction with code generation is novel. We have also included the sorting in the tree (moving more frequent nodes to the top) in the same way frequent reactions are moved to the beginning of the array in the optimized direct method. Since the number of reactions is static, dynamic removal and addition to the tree, as we had to consider in the past [26], is not an issue.

6 RESULTS

In this section, we analyze the performance results from our simulator implementations. We first look into the substeps of the algorithm and measure how much partial evaluation and reaction sorting reduces the time cost of these substeps. This gives a first idea of how different models benefit differently from these optimizations. To gain an even more precise understanding of these effects, we then measure the overall runtime of several benchmark models with different degrees of optimization. In addition, we investigate the influence of some further optimizations on the performance of our approach.

All measurements were done on a Dell PowerEdge R730 Server using an Intel Xeon E5-2683 v4 CPU. Hyperthreading was deactivated, and the machine was only executing the benchmark. We limited the maximum CPU frequency to 2.1 GHz by deactivating all Turbo Boost functionality. On the software side, the machine was running Linux 3.10 on CentOS 7. We installed the lasted version of all compilers for the performance measurements. We used Python 3.8.1, GCC 9.2.0, clang 9.0.1, and the Rust ecosystem at 1.41.0. The compilation was always done using maximum optimization, that is, using `--release` for Rust and `-O3` for the C++ compilers. Also link-time optimization (`-LTO`) (where indicated) and machine specific code (`--march=native`) was used. A Python script to recreate all measurements, figures, and tables from this paper is provided alongside the other software in the supplementary material.

As benchmarks for the different types of simulators and their configurations, we use the five models presented in the performance review by Gupta and Mendes [16]. Due to the subdivision of the generated code into individual files (Section 5.2), our implementation handles even the largest of those benchmark models (in contrast to an earlier version of our code [27]). In addition to the benchmark models, we added a stochastic variant of the Lotka-Volterra prey-predator model to serve as an example for an extremely small model. Table 1 lists all models with their specific characteristics, including the number of lines of code generated and the size of the resulting binary executable.

6.1 Microbenchmarks of the Substeps

Measuring the overall runtime of a simulator as a benchmark is relatively straightforward. We are, however, interested in understanding the performance profile in detail and therefore need more precise data than just total runtime or time per step. We thus created a set of microbenchmarks to measure the three phases (substeps) performed in each iteration of the algorithm:

Table 1. Overview of the Different Models used for the Performance Measurements

model	degree	species	reactions	LoC	Size of executable [MB]
Lotka-Voltera	2	2	3	~300	0.78
Multistate	7	9	18	~800	0.81
Multisite	55	66	288	~23 000	1.2
EGFR	163	356	3 749	~800 000	7.5
BCR	381	1 122	24 280 (24 388)	~9 800 000	69
FcϵRI	1 227	3 744	51 136 (58 276)	~64 000 000	350

Also displayed are the **lines of code (LoC)** of the generated specialized simulator as well as the number of reactions and species in the model and the size of the binary of the specialized simulator. The LoC are roughly similar for the two target languages C++ and Rust. The *degree* of the model is the average number of reactions that need to be updated when executing a reaction. The average is taken over the set of reactions and not weighted with their propensities. Given in parentheses is the number of reactions found (and used) by BioNetGen. The difference is because some reactions have a rate of zero and are therefore automatically excluded from our simulator.

- *random number generation* (line 6 and line 11 of Algorithm 1)
- *reaction selection* (line 7 of Algorithm 1)
- *reaction execution* (line 14 and line 3 of Algorithm 1) via the dependency graph as described in Section 4.2.

These substeps are on the order of 10 to 1,000 nanoseconds and, thus, hard to measure precisely. It is common practice to measure such fast functions in microbenchmarks by calling the function n times and measuring the total runtime. For usual numerical functions, this works well, but our three phases require a specific state of the simulator as input and, in the case of reaction execution, even alter that state. For this microbenchmarking, we use the Criterion.rs Library,[3] which provides methods and statistics for performing microbenchmarks. For the microbenchmarks, we employ the Rust variant of the specialized simulator.

To measure the desired quantities, we use a scheme that varies the number of replications, where in each replication, we perform 1,024 steps with an already warmed-up simulator. The replications are executed sequentially and use the same initial state. The idea is then to increment the number of replications until the overall runtime depends linearly on the number of replications (i.e., the domain beyond warm-up and measurement overhead). The slope of that linear dependency is then the runtime per replication.

The next challenge is to measure the time needed for the three different phases of each simulation step, as mentioned above, instead of only the time that the entire simulation step needs. To achieve this, we compare the runtimes taken for modified variants of the simulation algorithm. Figure 2 illustrates these modifications. We use two different methods to get the needed substep resolution.

First, we use an additive method, where extra work is done per step. Subtracting the time of the regular step gives us an estimate of the extra work. Instead of two random numbers, we generate four, and instead of selecting one reaction to fire we select two (and then only fire one). This gives us an estimate of the time for the selection and the random number procedure, with execution taking up the remainder of the time. Of course, we cannot execute the reaction twice, as this would lead to incorrect behavior, but we can assume the execution time to be the remaining time of the step.

[3] Criterion.rs is found at https://github.com/bheisler/criterion.rs.

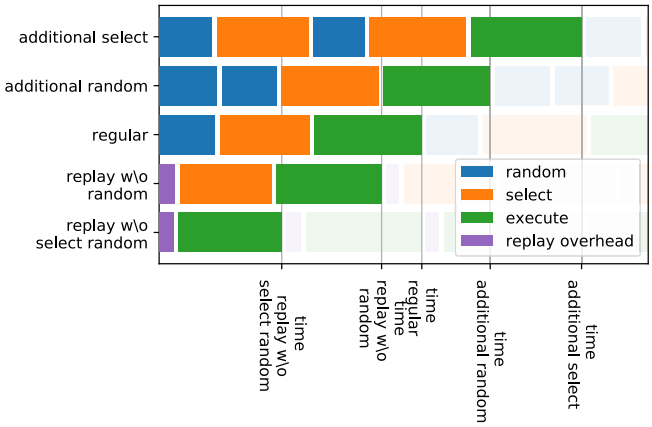


Fig. 2. An illustration of the different benchmark loops. Based on a measurement for the total time of a loop iteration (computed by running many iterations and dividing the time needed by the number of iterations), we can get an estimate for the different components that make up the loop, by adding and subtracting individual components, for example an estimate of t_{random} would be $t_{\text{additional random}} - t_{\text{regular}}$.

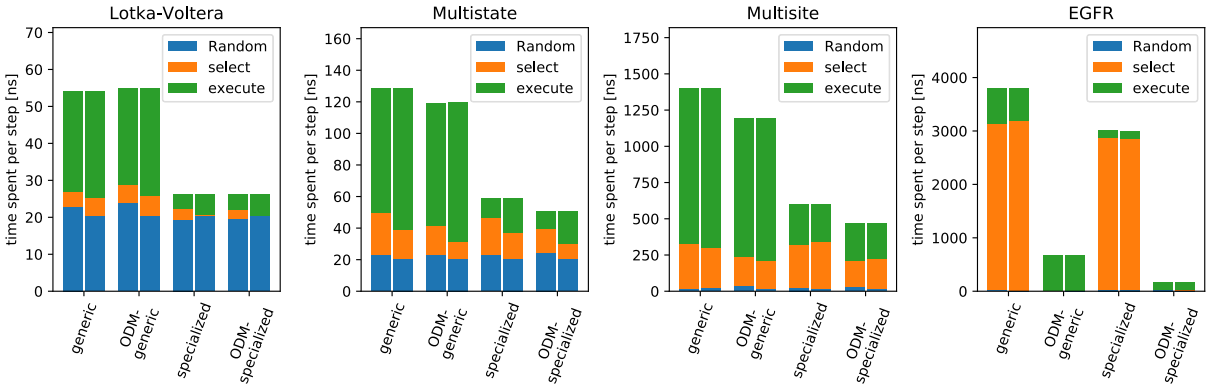


Fig. 3. Composition of the runtime of differently sized models, in both the specialized and the generic variant. The models are ordered left to right by the model size. The code generation predominantly affects reaction execution. For larger models, the effect of order optimization is also visible as reaction selection time is reduced. The two bars of measurement show the two ways of profiling the functions (Section 6.1). The left one is the additive, and the right one the replay approach. Both are written in Rust.

In the second approach, we execute the steps first in the regular way and store what random numbers are generated and what reactions are executed. Now we measure the time it takes to execute the steps, but without explicitly calculating random numbers or selecting a reaction. Instead, the stored values are used. This is essentially a replay of the simulation run. We try to get an estimate of the overhead due to replay. Therefore, we run the simulator regularly (including creating random numbers, etc.) and still use the replay numbers. The difference between this run to the regular run should give an estimate of the replay overhead. The replay method has trouble resolving short times for the select operation. In general, however, both methods result in largely similar numbers.

The results of the measurement setup, as described in the previous section, can be found in Figure 3. There are several conclusions we can draw from the data. First of all, smaller models are significantly influenced by random number generation, whereas random number generation has little effect on larger models. This is because random number generation takes on the order of 20 ns, independent of the size of the model.

Table 2. Different Levels of Code Generation

Simulator	reaction effect	reaction rate	dependency graph
generic	dynamic	dynamic	dynamic
Intermediate level **1**	evaluated	dynamic	dynamic
Intermediate level **2**	evaluated	evaluated	dynamic
Fully **S**pecialized	evaluated	evaluated	evaluated

The bold letters indicate the corresponding markers in Figure 4.

The time needed for random number generation is nearly the same for all simulators, since all use the same random number generator. We notice that for the case of the extremely small model (the Lotka-Volterra model), the time for the random number generation slightly decreases in the specialized simulator. This could potentially be due to less overall simulation effort being spent elsewhere, resulting in more of the random number generator staying in the cache.

Reaction execution is affected the most by specialization. When executing a reaction, the generic simulator needs to do a lot of work to look up what to do, for example what species to alter and what rate constant a reaction has. Furthermore, the dependent reactions also need to be updated. The specialized simulator has all this hard-coded in the assembly. The observed speedup in the execution phase is around 6× for the models tested.

Reaction selection only plays a minor role for smaller models. On the other hand, for the larger models we tested, it can be the dominating factor in runtime performance. The reaction sorting as done by the ODM can provide speedup of up to two orders of magnitude for the selection phase of the largest model tested.

6.2 Overall Performance

The previous section shows that optimizing the reaction ordering as well as partial evaluation can significantly reduce the simulation runtime. Now we investigate for *which parts of the model* partial evaluation has the largest effect. Therefore, we distinguish and compare four different degrees of partial evaluation (Table 2).

- The most basic variant is the generic simulator. Here no performance-relevant code generation does take place. Only for the purpose of simplifying the implementation, we perform code generation for some non-performance-critical parts like IO.
- At the intermediate level 1 the reaction effects (changes to the populations in the system) are partially evaluated.
- At the next intermediate level (intermediate level 2) additionally, the rate calculation is partially evaluated.
- Finally in the fully specialized variant the dependencies between the reactions are also hard-coded.

The reasonable expectation here is that with an increasing degree of partial evaluation, the compile-time increases, whereas runtime decreases. The key results are shown in Figure 4. Here we see how the compile time on the x-axis and the runtime (for simulation until $t = 100$ as was done in the original paper [27], which this paper is based on).

The marker shape designates how much partial evaluation was performed in terms of code generation. Here **S** and **g**, refer to the specialized and generic simulators and **1** and **2** to the intermediate variants. A tilted marker indicates that we have additionally done a pilot run to sort the reactions resulting in the algorithm being the **Optimized Direct Method (ODM)**. The color indicates how

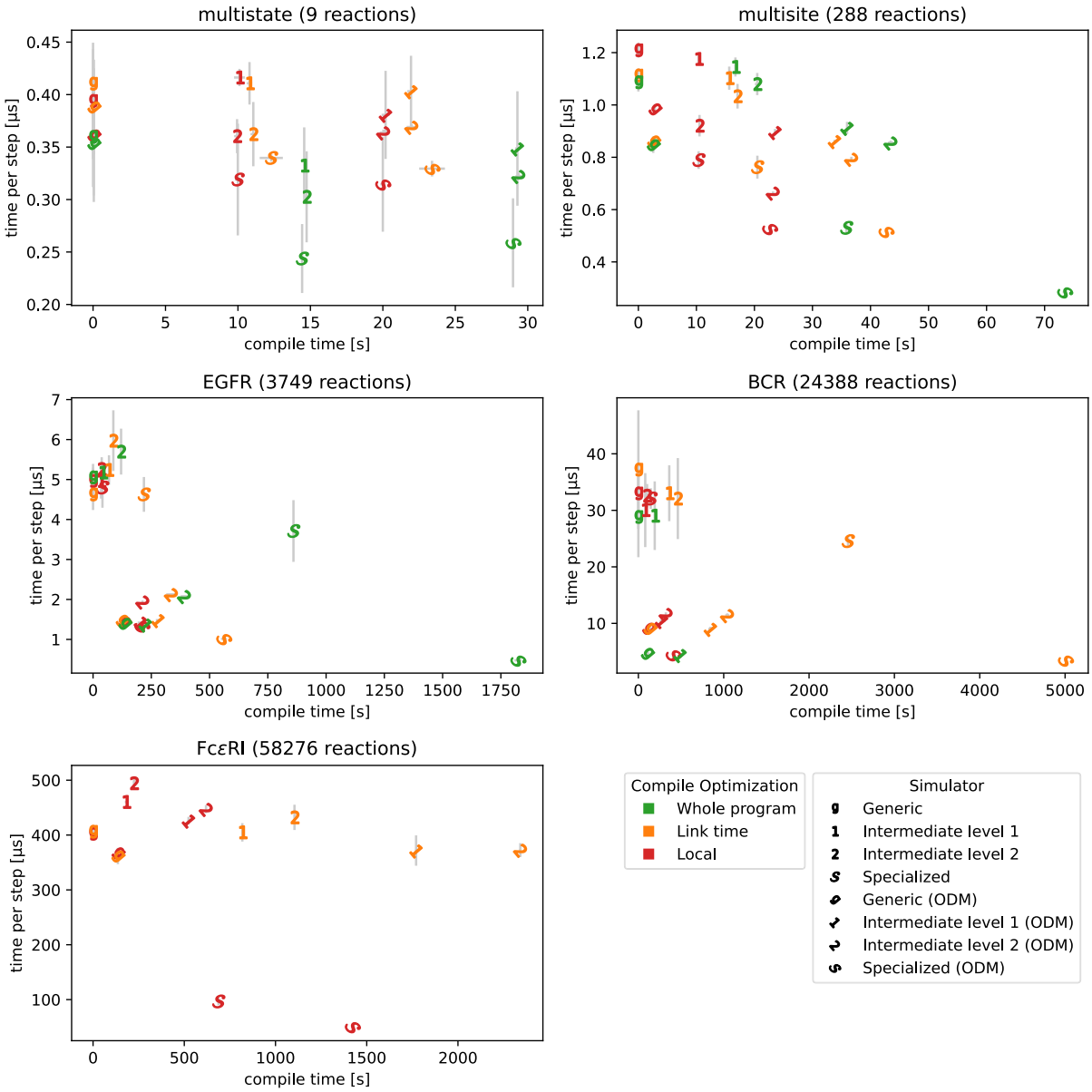


Fig. 4. Data for the Cost/Benefit ratio of different evaluation strategies. Error bars indicate one standard deviation. Data differs somewhat from what is shown in Figure 3, as time per step is not explicitly measured, but taking from the entire model run, including other IO, etc. Also model specific behavior is different, as a different simulation time is used. In general, we observe a Pareto-type distribution, with mostly diminishing returns with increasing compile time. For example, for the EGFR model, the change from link time (orange) to whole program (green) optimization significantly increases compile-time cost but only slightly decreases runtime for the specialized simulator ($\mathcal{S}$).

far-reaching the used compiler optimization was. The options are red, where optimizations were applied locally to each compilation unit, orange, where some link-time optimization was used, and green, where the whole program was compiled as a single unit, allowing the compiler to use all options of whole program optimization. The panels are sorted by the size of the model shown.

We would expect to see a Pareto-type distribution of the values in the plot. For those markers further left, we invested relatively little (or no) compile time. We would expect those to be slower (further up in the plot). For those markers further right in the plot, we spent more time on the

partial evaluation. If partial evaluation decreased the execution time, we would expect those markers to be further down.

We will discuss these results in three separate sections. First, we consider the model execution time. Next, we discuss the compilation time or the overhead for partial evaluation. Finally, we discuss how these two relate and where the tradeoff is.

6.2.1 Model Execution. We observe different behavior of model execution when considering the different models. As already seen in the profiling section, for the multistate model, sorting of reactions provides no advantage. This model has a short overall execution time. Therefore there is a fairly large discrepancy between the reported CPU time and the measured total time. The CPU time does not include parsing of the input file (where applicable) and writing of the output file. In addition, there is a language-specific overhead, as even native languages have a small startup time. This overhead is different between C++ and Rust, with Rust reporting the shortest CPU time but taking the longest to actually execute the model. For larger models, this overhead makes little difference.

What we can observe here nicely is that runtime decreases by a constant factor with every level of partial evaluation. (the different colors). They share a compile-time (discussed below) but have different execution times. Link-time optimization has no measurable effect for this small model. Moving the whole program to a single unit of execution, however, reduces execution time by about 20%.

The multisite model is primarily dominated by reaction execution time. Therefore, the specialized variants have an advantage here. BioNetGen is comparatively slow for this model. This is due to BioNetGen's simulation algorithm, which reorders reactions after every execution. This potentially reduces reaction selection time but introduces some additional overhead when executing a reaction. In regard to the compilation options and level of partial specialization, we have similar results to the multistate model.

The advantages of BioNetGen's Sorting Direct Method can be observed for the EGFR model. Here, simulation time is dominated by reaction selection for the generic simulators. Therefore, all simulators, generic or specialized, are roughly equally fast. Sorting after taking a model profile, however, greatly reduces total execution time. In general, for this model, sorting is significantly more impactful than specialization. However, in combination, both of these together can reach very high performance. A big driver for the performance of the EGFR model is the order of the reactions, as a few reactions dominate the execution. Therefore, we have a large variation in the execution time for the randomly ordered simulations and a low error bar for the sorted variant.

Similar to the BCR Model, reaction sorting dominates. For this model, however, full code generation into a single file is no longer tractable. We have put an arbitrary limit here of a compile-time of more than one hour. Even the best specialized variant (here: link-time optimization, sorted) performs similarly to the best generic simulator and only slightly better than without link-time optimization, but with a large compile overhead.

The FcϵRI model, which is larger yet again, has a more uniform distribution of the reactions. Therefore, reaction sorting mostly results in a minor improvement in performance. Specialization, however, when done in the form of full code generation, reduces runtime threefold. And again, we can observe how sorting, once all other overhead is reduced, has a significantly larger impact on total runtime performance.

There is another interesting artifact. We can observe for the two larger models (BCR and FcϵRI) that the partial specialization actually produces runtimes that are worse than the generic simulator. We attribute this to the code size of the generated binary. Whereas the generic simulator has fairly minimal code size, the partially specialized ones have at least a few functions for every reaction.

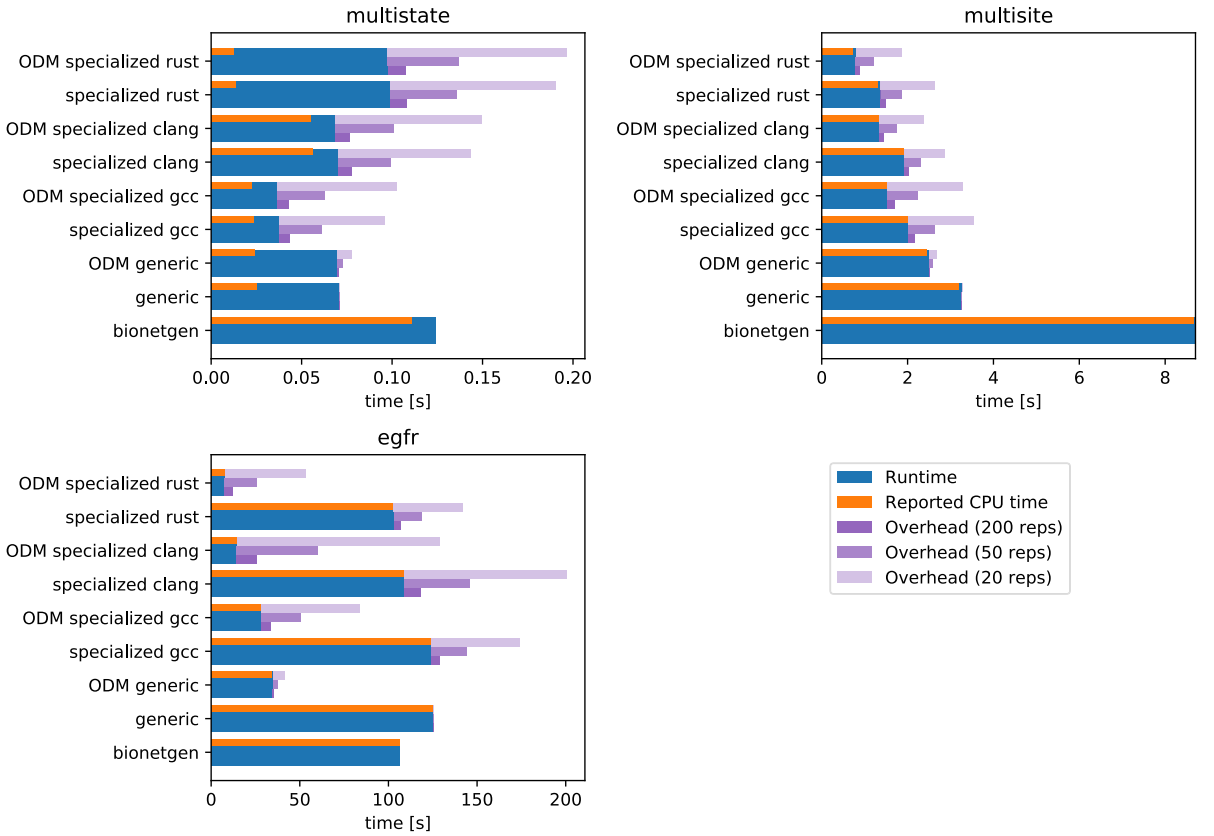


Fig. 5. This figure compares the runtime of different code generation targets (i.e., compilers and languages) across the three smaller models. The runtime of the different simulators for a single simulation run is shown in blue. Optimizing the order of reactions has a larger effect for larger models. Shown in purple is the exemplary amortized overhead (i.e., the overhead per simulation run) for a different number of replications. This overhead includes compile-time, time to parse the model input, and time to perform the profile-guided optimization, where applicable. We see that for few replications, partial evaluation is not beneficial, but for more replications, the overhead per simulation run vanishes.

Therefore, there is some overhead when executing a reaction because the code needs to be loaded. If the simulator is only partially specialized, this outweighs the improvement in performance due to specialization.

From the different degrees of specialization, we can also get insight into where the partial evaluation has the largest effect. The first intermediate level (**1**) has little effect (and, as discussed above, can even degrade performance). This is expected, as the pure effect of reaction execution takes up a relatively minor part of total runtime. The largest step up in performance comes from the partial evaluation of the dynamic dependencies. We can generally observe that when the dependency graph is not partially evaluated, the separation in different units of compilation has relatively little effect. This implies that part of the speedup is due to the compiler optimizing the hard-coded interaction between reaction execution and dependencies between reactions.

6.2.2 Overhead of Compilation. In Section 2.1 we already mentioned that there is overhead associated with the partial evaluation α. In our approach, this overhead depends on the actual compiler used to compile the specialized simulator. Figure 5 shows the results for the full specialization but for different compiler setups. Three different compilers for the generated specialized code are considered. We only use the smaller models here, as full specialization is not tractable for the larger models.

- The GNU **Compiler Collection (GCC)** is a well-established compiler suite.
- Clang is a C/C++ frontend for the LLVM compiler. It provides a more modern approach but is not as established as GCC.
- The Rust compiler Rustc is also based on an LLVM backend.

In addition to the total runtime of the simulator for the respective model, we have also measured the overhead incurred by both compilation and profile-guided optimization.

There is no clearly fastest compiler when it comes to the compile time of the models. Nonetheless, the compile times differ by a factor of 1.4 to 2×, depending on the model. However, with every model, another compiler is consistently the fastest. This relation is certainly an interesting avenue for future exploration.

In Figure 4 we can observe how partial specialization affects the overhead. For the smaller models, in particular, the multistate model, the general framework, like the output routines and the simulation loop, dominate compile time. Therefore, compile time is only affected by what compile settings were used. Interestingly, for the multistate model with only nine reactions, the full Specialized Simulator consistently compiles slightly faster than the partially specialized simulators. This is because compiling the code needed to dynamically load this information takes slightly more time than just hard coding them due to the involvement of IO libraries, etc. This only holds true for small models. For larger models, we can clearly see how the different degrees of code generation affect compile time.

The compiler settings are effectively a measure of the parallelization during compilation. Since the machine used for the benchmark is a parallel-execution-capable server, the traditional compilation, which is then split into several files and linking, remains fairly tractable even for larger models.

We can also observe different compile times for the different levels of partial evaluation. There is little difference in compile time between intermediate level 1 and intermediate level 2. This is because the majority of the generated code is part of the dependency graph, which neither of them partially evaluates.

It should be noted that other than time performance, memory performance can also be a critical limiter for the code generation presented. Specifically, the compilation of extremely large source code files is rather challenging for the compilers. We did not conduct a detailed study on the memory usage, but with the larger models, we not only ran into runtime limitations but also the limits of the memory on our 32 GB RAM desktop workstation. Therefore we ran everything on a larger server.

6.2.3 Runtime-compiletime Tradeoff. Regarding the overall overhead of compilation, there is a tradeoff. The deciding factor is the number of replications that are executed. Stochastic models need to be executed multiple times to get statistically meaningful results. When executing hundreds or thousands of runs, the amortized compilation overhead is minimal (cf. Figure 5). For only a handful of runs, however, due to the overhead, partial evaluation appears hardly worth it.

It is almost always most meaningful to either go for full specialization or just use a generic simulator with the Optimized Direct Method (i.e., with a pilot run used for sorting). The intermediate solutions do not appear to perform particularly well in comparison. The yield of generating a simulator is somewhat model-dependent. In all cases, the specialized simulator clearly outperforms the generic one. On the other hand, especially for the larger models, the overhead is significant. For the BCR model, for example, a slightly faster simulator is achieved, but only at high compile-time costs in the 1000s of seconds. Likely an even faster simulator could have been created if more compilation time would have been used (more than an hour), but only many replications (1000s) would justify this investment. For the FcϵRI model, specialization was significant, even without

any link-time optimization. At this point, we have no indicator or automatic way to estimate the likelihood of success when performing partial evaluation.

6.3 Performance Impact of other Optimizations

After precisely profiling the simulation algorithm and illuminating the interplay between partial evaluation and profile-guided optimization, we now investigate some further factors that impact the performance.

6.3.1 Dynamic Parameters. As described in Section 5.3, some parameters may be specified to be read in at runtime. To measure this, instead of hard coding the reaction rate constants, we have instead modified the models such that each rate expression requires one constant to be evaluated. Depending on the model, we see a performance decrease between 1% and 7%. The larger models have more of a performance penalty for two reasons. Firstly, as we have seen above, for these models, rate calculation is a more dominant part of the overall execution time. Therefore, the slowdown in calculation has more of an effect there. Secondly, for smaller models, we can expect more of the model to remain in the cache. Therefore, reading the parameter value should be less costly.

6.3.2 Tree-based Propensity Calculation. The reaction selection phase can be the dominant part of the runtime. This is especially the case for the unsorted variants of the simulators. In those cases, we have seen a significant performance improvement when using the tree-based reaction selection as described in Section 5.4.2. Instead of linear complexity in selecting the reaction, the selection now has logarithmic complexity. For some models, like the unsorted EGFR model, we, therefore, see up to a 40% reduction in total execution time. On the other hand, once reactions are sorted, we actually see a 40% increase in runtime. That is because there is an overhead to keeping this tree data structure when modifying a reaction rate. We conclude that using this tree structure makes sense for large systems, where the dominant reactions have not been identified (and moved first), either because no pilot run was done or because the model behavior does not have predictable dominant reactions.

The tree-based reaction selection works well in conjunction with the partial evaluation of reaction rate expressions. Similar to reaction sorting, because partial evaluation reduces rate calculation time so drastically, changes to reaction selection time have a large effect. This exaggerates the effect of the reaction selection runtime. However, tree-based propensity calculation appears to be an inferior choice to reaction sorting for the tested models. Therefore, in cases where partial evaluation makes sense, the overhead of the pilot run usually amortizes, and we see no need for the tree data structure. Nonetheless, there might be cases where such an approach might be useful.

6.4 Applications of Code-generation for Simulation

Code generation could be of interest to the simulation community beyond the scope of pure performance improvements. One application for which we have used our tool is generating code for teaching. With only about an hour's work, it was possible to provide another output target in the form of a Python library. We could then supply this as a self-contained model to students, for the purpose of teaching model experimentation techniques, like parameter estimation and optimization. This implementation is not very fast, however sufficient for the purpose of teaching and could be easily integrated with existing language frameworks.

A second application where code generation has been used in other domains [2] is to create web tools. Based on the Rust version of the code, we have generated a webassembly variant of the code. The use case here is to supply a link to a webpage alongside a published model. On this webpage, some basic experiments could be performed by interested parties with reasonable performance.

7 DISCUSSION & FUTURE WORK

In this paper, we have explored two ways to speed up the stochastic simulation of biochemical reaction networks. Based on the well-established Direct Method, we investigated partial evaluation/program specialization via code generation as well as profile-guided optimization in the form of the Optimized Direct Method. Both techniques target different steps in the Direct Method. Whereas partial evaluation mainly improves the performance of reaction execution, profile-guided optimization provides speedup mostly to reaction selection. We illustrated these dependencies by carefully examining the performance of different simulator variants for a set of realistic benchmark models. With only profile-guided optimization, we achieved a total speedup of 4× for the EGFR model, whereas partial evaluation alone provided a speedup of up to factor only 20%. With both techniques combined, however, we measured a speedup of up to a factor of 16×. Since either technique targets a different phase of the simulation algorithm, they work great in conjunction with one another.

When comparing with the state of the art as surveyed by a recent review [16], our generated simulators outperform all other implementations in the models we tested. However, our approach has some limitations. First, both partial evaluation and profile-guided optimization create computational overhead, which is typically justified if many runs are executed. Executing only a single run, however, can usually not benefit from our approach. Second, when applying the profile-guided optimization, we assume that a single instrumented run is representative of all further runs of the same model with the same parameters. In our experience, as well as the observations made in the literature [4], this appears to be a valid assumption for most models. Third, as discussed below, generating specialized simulators for very large models requires some compromises on the depth of partial evaluation, meaning that in such cases, a specialized simulator is not necessarily faster than a generic simulator.

Neither the optimal ordering of reactions by profile-guided optimization nor the partial evaluation of SSA variants is novel. However, as our measurements show, the combination of both techniques has the potential to produce highly efficient algorithms. The key to this performance gain is specializing the algorithm using static and dynamic properties of the model. In the first step, partial evaluation exploits information obtained by static analysis of the model to generate optimized, model-specific algorithms, for example by removing any kind of indirection in the execution, especially when evaluating rate expressions or following the dependency graph. In the second step, profile-guided optimization helps to further specialize the simulator based on dynamic information obtained from a pilot run (or several ones), for example by optimally ordering reactions.

In this paper, we also performed some in-depth performance analyses of the specialized simulators. For the smaller models, we profiled the time spent in the different phases of the algorithm and could thereby find out how the different optimizations affect the phases. Specifically, how partial evaluation reduces the time for reaction execution and reaction sorting can reduce reaction selection time. We also varied the amount of partial evaluation performed by introducing different levels of partial evaluation. Our results here mainly indicate that it is usually not worth it. The lower levels of partial evaluation provide only little practical use. We did, however, get the insight that especially the partial evaluation of the dependency graph is what explains the performance of the specialized simulator.

When considering large models, generating a single file of source code is impractical due to large compilation time and memory needs. We, therefore, have explored strategies to mitigate this. We found separation into different compilation units to be helpful at improving compilation time when compiling on a parallel machine. Link-time optimization provides an intermediate solution

that somewhat increases compilation time but also improves execution time. Nonetheless, the performance of these faster to compile variants is not as great as for the single-file variant because reaction execution now stretches (potentially) over many compilation units. In most cases, it will, however, still beat a generic simulator, but the tradeoff is not as clear. A contributing factor here also becomes code size, as for the larger models, large binaries are generated, compared to the relatively compact generic simulator.

There are two software artifacts associated with this study. Firstly, the same scripts we used to create all the graphs and data for this paper are available at https://git.informatik.uni-rostock.de/mosi/ssa_code_generator. Furthermore, as a starting point for anybody interested in trying out code generation for SSA models, we provide a more basic but usable implementation at https://git.informatik.uni-rostock.de/mosi/compact-ssa-code-gen.

In most standard SSA approaches, the reactions follow mass-action kinetics. This means that a reaction's propensity (the likelihood of it happening) is calculated by multiplying a constant (the rate constant) with the amounts of the input species of the reaction. Some tools support arbitrary rate functions which are applied to the current state. These arbitrary rates pose a performance problem for generic simulators because every time they are calculated, the expression tree needs to be re-evaluated. For our partial evaluation approach, this is of less concern because these functions are compiled similarly to the mass action kinetics. This benefit can be used with only little implementation effort. In [31], a similar strategy was adopted to generate byte code for rate expressions, but via Java runtime reflection.

Our results motivate future work in different directions.

The limitations of our approach are closely linked to the underlying compiler technology. While faster computation, as in faster memory and CPUs, would decrease the overhead time for partial evaluation, the larger gain is to be made within the algorithmic structure of the compiler. Ideally, the compiler would detect the repeated patterns in the reaction rules and avoid redundant work. Future work could also include finding ways to hint this to the compiler.

Further future work will focus on the generation and compilation of source code on the one hand and on the actual simulation algorithm on the other hand.

Compilation Improvements. For the profile-guided optimization, we could also employ the profiling options of the compiler itself, where after running an instrumented binary, it generates more specific optimized code. The optimizations used there are on a smaller scale than the reaction sorting done for this work, but other than an additional compilation, they are relatively easy to include and are certainly worth testing out. In addition to sorting the reactions, it could be useful to also sort the reactants by usage frequency. Having the most accessed species located next to one another in memory should improve cache locality.

A final improvement could be to explore the possibilities of minimizing the linking against system libraries for the generated simulators. Specifically, since memory allocation and error handling is not needed, linking against such capabilities only increases startup time. This is especially noticeable for Rust, where startup time is quite significant.

The profile-guided optimization is just one example of runtime-based optimization. It would be interesting to explore other runtime reconfiguration approaches [19] for use in conjunction with the partial evaluation.

It would also be interesting to explore the optimizations made by the compiler to the simulator structure. Potentially this could provide insight even for traditional generic simulators. Additionally, from a compiler engineering perspective, it would be interesting to explore how the assembly code generated by the different compilers differs. This could give potential insight into how to further improve compilers.

One side effect of separating the generated code into separate units is that it enables incremental compilation. That means that for a small change in the model, the entire model is not again evaluated. Instead, only the modified parts are re-compiled. However, some challenges exist to apply this approach. First, reactions and reactants are accessed via indices. When a rule or species is removed from the model, the consecutive ordering of reactions needs to be adjusted. This could be circumvented, for example, by not using arrays as structures but individual variables for the different species. However, this would drastically increase compile time. Second, changes to one reaction are not as local as one might expect when the dependency graph is used. Since the effects of reactions are all hardcoded, changing one reaction also means changing all dependent ones. Without the dependency graph, incremental compilation would be much more feasible, but also runtime performance a lot worse. Therefore, after initial tests, we decided against further pursuing incremental compilation.

Implications. In conclusion, we hope that future publications in the field of stochastic algorithms for reaction networks will consider the applicability of their algorithms to partial evaluation. Potentially algorithms in the past were designed to circumvent specific limitations of generic simulators. For example, some of them, such as the rejection-based method [42] (or also the approximative algorithm τ-leaping [14]), have been developed to reduce the effort needed for reaction execution. However, our code generation approach massively speeds up reaction execution, making this less of an issue. Additionally, we might find that algorithms previously considered slow could be "revived" because they benefit more from code generation than algorithms currently considered faster. Additionally, we hope that this work provides another argument for the use of domain-specific languages and a clear separation of model and simulation.

REFERENCES

[1] Pierre Boutillier, Mutaamba Maasha, Xing Li, Héctor F. Medina-Abarca, Jean Krivine, Jérôme Feret, Ioana Cristescu, Angus G. Forbes, and Walter Fontana. 2018. The Kappa platform for rule-based modeling. *Bioinformatics* 34, 13 (06 2018), i583–i592. https://doi.org/10.1093/bioinformatics/bty272 arXiv:https://academic.oup.com/bioinformatics/article-pdf/34/13/i583/25098638/bty272.pdf.

[2] Alessandro Bozzon, Sara Comai, Piero Fraternali, and Giovanni Toffetti Carughi. 2006. Conceptual modeling and code generation for rich internet applications. In *Proceedings of the 6th International Conference on Web Engineering (ICWE'06)*. Association for Computing Machinery, New York, NY, USA, 353–360. https://doi.org/10.1145/1145581.1145649

[3] R. Campos-Rebelo, F. Pereira, F. Moutinho, and L. Gomes. 2011. From IOPT petri nets to C: An automatic code generator tool. In *2011 9th IEEE International Conference on Industrial Informatics*. 390–395. https://doi.org/10.1109/INDIN.2011.6034908

[4] Yang Cao, Hong Li, and Linda Petzold. 2004. Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. *The Journal of Chemical Physics* 121, 9 (2004), 4059–4067. https://doi.org/10.1063/1.1778376 arXiv:https://doi.org/10.1063/1.1778376

[5] Martin Carlsson. 2018. *Automatic Code Generation from a Colored Petri Net Specification for Game Development with Unity3D.*

[6] Stefano Carnà. 2020. Reproducibility report for the paper: 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 (SIGSIM-PADS'20)*. Association for Computing Machinery, New York, NY, USA, 171–173. https://doi.org/10.1145/3384441.3396228

[7] Deniz Çetinkaya, Alexander Verbraeck, and Mamadou D. Seck. 2015. Model continuity in discrete event simulation: A framework for model-driven development of simulation models. *ACM Trans. Model. Comput. Simul.* 25, 3, Article 17 (April 2015), 24 pages. https://doi.org/10.1145/2699714

[8] Daniel D. Deavours, G. Clark, T. Courtney, D. Daly, Salem Derisavi, J. Doyle, W. Sanders, and P. Webster. 2002. The Möbius framework and its implementation. *IEEE Trans. Software Eng.* 28 (2002), 956–969.

[9] PAH Dijk. 2018. *An Efficient Simulation of Crosslinked RAFT Copolymerization.* Master's thesis. University of Twente.

[10] James R. Faeder, Michael L . Blinov, and William S. Hlavacek. 2009. Rule-based modeling of biochemical systems with BioNetGen. In *Systems Biology*. Springer, 113–167.

[11] Peter M. Fenwick. 1994. A new data structure for cumulative frequency tables. *Software: Practice and Experience* 24, 3 (1994), 327–336.

[12] Yoshihiko Futamura. 1999. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher Order Symbol. Comput.* 12, 4 (Dec. 1999), 381–391. https://doi.org/10.1023/A:1010095604496

[13] M. A. Gibson and J. Bruck. 2000. Efficient exact stochastic simulation of chemical systems with many species and many channels. *Journal of Physical Chemistry A* 104, 9 (2000), 1876–1889. http://pubs3.acs.org/acs/journals/doilookup?in_doi=10.1021/jp993732q.

[14] Daniel Gillespie. 2001. Approximate accelerated stochastic simulation of chemically reacting systems. *Journal of Chemical Physics* 115 (07 2001), 1716–1733. https://doi.org/10.1063/1.1378322

[15] Daniel T. Gillespie. 1977. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry* 81, 25 (1977), 2340–2361.

[16] Abhishekh Gupta and Pedro Mendes. 2018. An overview of network-based and free approaches for stochastic simulation of biochemical systems. *Computation* 6, 1 (2018). https://doi.org/10.3390/computation6010009

[17] Leonard A. Harris, Justin S. Hogg, José-Juan Tapia, John A. P. Sekar, Sanjana Gupta, Ilya Korsunsky, Arshi Arora, Dipak Barua, Robert P. Sheehan, and James R. Faeder. 2016. BioNetGen 2.2: Advances in rule-based modeling. *Bioinformatics* 32, 21 (07 2016), 3366–3368. https://doi.org/10.1093/bioinformatics/btw469 arXiv:https://academic.oup.com/bioinformatics/article-pdf/32/21/3366/7889722/btw469.pdf

[18] Monika Heiner, Mostafa Herajy, Fei Liu, Christian Rohr, and Martin Schwarick. 2012. Snoopy–a unifying petri net tool. In *International Conference on Application and Theory of Petri Nets and Concurrency*. Springer, 398–407.

[19] Tobias Helms, Roland Ewald, Stefan Rybacki, and Adelinde M. Uhrmacher. 2015. Automatic runtime adaptation for component-based simulation algorithms. *ACM Trans. Model. Comput. Simul.* 26, 1, Article 7 (Oct. 2015), 24 pages. https://doi.org/10.1145/2821509

[20] Tobias Helms, Tom Warnke, Carsten Maus, and Adelinde M. Uhrmacher. 2017. Semantics and efficient simulation algorithms of an expressive multilevel modeling language. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 27, 2 (2017), 1–25.

[21] Jan Himmelspach and Adelinde M. Uhrmacher. 2007. Plug'n simulate. In *40th Annual Simulation Symposium (ANSS'07)*. IEEE, 137–143.

[22] Neil D. Jones. 1996. An introduction to partial evaluation. *ACM Comput. Surv.* 28, 3 (Sept. 1996), 480–503. https://doi.org/10.1145/243439.243447

[23] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2014. Combinators for impure yet hygienic code generation. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2543728.2543740

[24] Sarah M. Keating, Dagmar Waltemath, Matthias König, Fengkai Zhang, Andreas Dräger, Claudine Chaouiya, Frank T. Bergmann, Andrew Finney, Colin S. Gillespie, Tomáš Helikar, et al. 2020. SBML Level 3: An extensible format for the exchange and reuse of biological models. *Molecular Systems Biology* 16, 8 (2020), e9110.

[25] Gabriele Keller, Hugh Chaffey-Millar, Manuel M. T. Chakravarty, Don Stewart, and Christopher Barner-Kowollik. 2008. Specialising simulator generators for high-performance Monte-Carlo methods. In *Practical Aspects of Declarative Languages*, Paul Hudak and David S. Warren (Eds.). Springer Berlin„ 116–132.

[26] Till Köster and Adelinde M. Uhrmacher. 2018. Handling dynamic sets of reactions in stochastic simulation algorithms. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS'18)*. Association for Computing Machinery, New York, NY, USA, 161–164. https://doi.org/10.1145/3200921.3200943

[27] Till Köster, Tom Warnke, and Adelinde M. Uhrmacher. 2020. 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 (SIGSIM-PADS'20)*. Association for Computing Machinery, New York, NY, USA, 159–170. https://doi.org/10.1145/3384441.3395983

[28] Roland Leissa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A partial evaluation framework for programming high-performance libraries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 119 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276489

[29] Mieszko Lis, Maxim N. Artyomov, Srinivas Devadas, and Arup K. Chakraborty. 2009. Efficient stochastic simulation of reaction-diffusion processes via direct compilation. *Bioinformatics* 25, 17 (07 2009), 2289–2291. https://doi.org/10.1093/bioinformatics/btp387 arXiv:https://academic.oup.com/bioinformatics/article-pdf/25/17/2289/16889395/btp387.pdf.

[30] James M. McCollum, Gregory D. Peterson, Chris D. Cox, Michael L. Simpson, and Nagiza F. Samatova. 2006. The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behavior. *Computational Biology and Chemistry* 30, 1 (2006), 39–49. https://doi.org/10.1016/j.compbiolchem.2005.10.007

[31] T. Meyer, T. Helms, T. Warnke, and A. M. Uhrmacher. 2018. Runtime code generation for interpreted domain-specific modeling languages. In *2018 Winter Simulation Conference (WSC)*. 605–615. https://doi.org/10.1109/WSC.2018.8632545

[32] Andrew K. Miller, Justin Marsh, Adam Reeve, Alan Garny, Randall Britten, Matt Halstead, Jonathan Cooper, David P. Nickerson, and Poul F. Nielsen. 2010. An overview of the CellML API and its implementation. *BMC Bioinformatics* 11, 1 (2010), 1–12.

[33] Melissa E. O'Neill. 2014. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Technical Report HMC-CS-2014-0905. Harvey Mudd College, Claremont, CA.

[34] Oleksandr Ostrenko, Pietro Incardona, Rajesh Ramaswamy, Lutz Brusch, and Ivo F. Sbalzarini. 2017. pSSAlib: The partial-propensity stochastic chemical network simulator. *PLOS Computational Biology* 13, 12 (12 2017), 1–15. https://doi.org/10.1371/journal.pcbi.1005865

[35] R. Pais, S. P. Barros, and L. Gomes. 2005. A tool for tailored code generation from Petri net models. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*. 857–864. https://doi.org/10.1109/ETFA.2005.1612615

[36] Isabel Cristina Pérez-Verona, Mirco Tribastone, and Andrea Vandin. 2019. A large-scale assessment of exact model reduction in the biomodels repository. In *Computational Methods in Systems Biology*, Luca Bortolussi and Guido Sanguinetti (Eds.). Springer International Publishing, Cham, 248–265.

[37] Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging. *Commun. ACM* 55, 6 (June 2012), 121. https://doi.org/10.1145/2184319.2184345

[38] Boris Yakovlevich Ryabko. 1989. A fast on-line code. In *Doklady Akademii Nauk*, Vol. 306. Russian Academy of Sciences, 548–552.

[39] Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. 2003. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems* 25, 4 (July 2003), 452–499. https://doi.org/10.1145/778559.778561

[40] Michael D. Smith. 2000. Overcoming the challenges to feedback-directed optimization (keynote talk). *SIGPLAN Not.* 35, 7 (Jan. 2000), 1–11. https://doi.org/10.1145/351403.351408

[41] Endre T. Somogyi, Jean-Marie Bouteiller, James A. Glazier, Matthias König, J. Kyle Medley, Maciej H. Swat, and Herbert M. Sauro. 2015. libRoadRunner: A high performance SBML simulation and analysis library. *Bioinformatics* 31, 20 (06 2015), 3315–3321. https://doi.org/10.1093/bioinformatics/btv363 arXiv:https://academic.oup.com/bioinformatics/article-pdf/31/20/3315/17087875/btv363.pdf.

[42] Vo Hong Thanh. 2018. A critical comparison of rejection-based algorithms for simulation of large biochemical reaction networks. *Bulletin of Mathematical Biology* 81, 8 (July 2018), 3053–3073. https://doi.org/10.1007/s11538-018-0462-y

[43] Yentl Van Tendeloo and Hans Vangheluwe. 2017. Classic DEVS modelling and simulation. In *Proceedings of the 2017 Winter Simulation Conference (WSC 2017)*. IEEE, 644–656. https://doi.org/10.1109/WSC.2017.8247822

[44] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. 2000. *Theory of Modeling and Simulation* (2nd ed.). Academic Press, Inc., Orlando, FL.

[45] Jia Zeng. 2007. *Partial Evaluation for Code Generation from Domain-Specific Languages*. Ph.D. Dissertation. https://doi.org/10.7916/d8xp7cst

Received December 2020; revised July 2021; accepted November 2021

[B] Potential based, spatial simulation of dynamically nested particles

Background To study cell biological phenomena which depend on diffusion, active transport processes, or the locations of species, modeling and simulation studies need to take space into account. To describe the system as a collection of discrete objects moving and interacting in continuous space, various particle-based reaction diffusion simulators for cell-biological system have been developed. So far the focus has been on particles as solid spheres or points. However, spatial dynamics might happen at different organizational levels, such as proteins, vesicles or cells with interrelated dynamics which requires spatial approaches that take this multi-levelness of cell biological systems into account.

Results Based on the perception of particles forming hollow spheres, ML-Force contributes to the family of particle-based simulation approaches: in addition to excluded volumes and forces, it also supports compartmental dynamics and relating dynamics between different organizational levels explicitly. Thereby, compartmental dynamics, e.g., particles entering and leaving other particles, and bimolecular reactions are modeled using pair-wise potentials (forces) and the Langevin equation. In addition, forces that act independently of other particles can be applied to direct the movement of particles. Attributes and the possibility to define arbitrary functions on particles, their attributes and content, to determine the results and kinetics of reactions add to the expressiveness of ML-Force. Its implementation comprises a rudimentary rule-based embedded domain-specific modeling language for specifying models and a simulator for executing models continuously. Applications inspired by cell biological models from literature, such as vesicle transport or yeast growth, show the value of the realized features. They facilitate capturing more complex spatial dynamics, such as the fission of compartments or the directed movement of particles, and enable the integration of non-spatial intra-compartmental dynamics as stochastic events.

Conclusions By handling all dynamics based on potentials (forces) and the Langevin equation, compartmental dynamics, such as dynamic nesting, fusion and fission of compartmental structures are handled continuously and are seamlessly integrated with traditional particle-based reaction-diffusion dynamics within the cell. Thereby, attributes and arbitrary functions allow to flexibly describe diverse spatial phenomena, and relate dynamics across organizational levels. Also they prove crucial in modeling intra-cellular or intra-compartmental dynamics in a non-spatial manner, and, thus, to abstract from spatial dynamics, on demand which increases the range of multi-compartmental processes that can be captured.

[270] T. Köster, P. Henning, and A. M. Uhrmacher, *Potential Based, Spatial Simulation of Dynamically Nested Particles.*, BMC Bioinformatics **20**, (2019)

METHODOLOGY ARTICLE Open Access

Potential based, spatial simulation of dynamically nested particles

Till Köster*, Philipp Henning and Adelinde M. Uhrmacher

Abstract

Background: To study cell biological phenomena which depend on diffusion, active transport processes, or the locations of species, modeling and simulation studies need to take space into account. To describe the system as a collection of discrete objects moving and interacting in continuous space, various particle-based reaction diffusion simulators for cell-biological system have been developed. So far the focus has been on particles as solid spheres or points. However, spatial dynamics might happen at different organizational levels, such as proteins, vesicles or cells with interrelated dynamics which requires spatial approaches that take this multi-levelness of cell biological systems into account.

Results: Based on the perception of particles forming hollow spheres, ML-Force contributes to the family of particle-based simulation approaches: in addition to excluded volumes and forces, it also supports compartmental dynamics and relating dynamics between different organizational levels explicitly. Thereby, compartmental dynamics, e.g., particles entering and leaving other particles, and bimolecular reactions are modeled using pair-wise potentials (forces) and the Langevin equation. In addition, forces that act independently of other particles can be applied to direct the movement of particles. Attributes and the possibility to define arbitrary functions on particles, their attributes and content, to determine the results and kinetics of reactions add to the expressiveness of ML-Force. Its implementation comprises a rudimentary rule-based embedded domain-specific modeling language for specifying models and a simulator for executing models continuously. Applications inspired by cell biological models from literature, such as vesicle transport or yeast growth, show the value of the realized features. They facilitate capturing more complex spatial dynamics, such as the fission of compartments or the directed movement of particles, and enable the integration of non-spatial intra-compartmental dynamics as stochastic events.

Conclusions: By handling all dynamics based on potentials (forces) and the Langevin equation, compartmental dynamics, such as dynamic nesting, fusion and fission of compartmental structures are handled continuously and are seamlessly integrated with traditional particle-based reaction-diffusion dynamics within the cell. Thereby, attributes and arbitrary functions allow to flexibly describe diverse spatial phenomena, and relate dynamics across organizational levels. Also they prove crucial in modeling intra-cellular or intra-compartmental dynamics in a non-spatial manner, and, thus, to abstract from spatial dynamics, on demand which increases the range of multi-compartmental processes that can be captured.

Keywords: Space, Simulation, Nesting, Force, Multi-level, Modeling, Attributed

Background

Space plays an important role in cell biological dynamics, such as cell signalling [1]. To study cell biological phenomena which depend on diffusion, active transport processes, or the locations of species, modeling and simulation studies need to take space into account. Over the last decade, a variety of spatial modeling and simulation methods and tools has been developed to support these simulation studies. They offer different approaches how to describe a spatial model, e.g., reaction-based, rule-based, or graphically, and they differ referring to what kind of spatial dynamics can be described, e.g., whether concentrations, populations, or individual particles are considered, whether a deterministic or stochastic approach is pursued, and whether movement takes place in discretized or continuous space [2, 3]. The kind of spatial

*Correspondence: till.koester@uni-rostock.de
[1]Institute of Computer Science, University of Rostock, Albert-Einstein-Straße 22, 18059 Rostock, Germany

dynamics that are supported by tools largely determines the results and questions that can be answered by a simulation study [4–6] and has been subject to different categorizations to structure the portfolio of spatial simulation methods from which the user can select.

TAKAHASHI et al. [7] distinguished spatial simulation approaches according to the representation of space. We adopt this structuring and slightly adapt it for the purpose of this paper as presented in Fig. 1. We distinguish between:

1. Stochastic non-spatial dynamics: To take the stochasticity of the modelled system into account, the propensity of reactions are sampled from exponential distributions and determine which reaction will occur at which time [9].
2. Compartmental dynamics: Many tools allow to constrain the dynamics of species to specific compartments. However also, compartments themselves can be subject to dynamics. For example, compartments can fuse and divide [10].
3. Individual particles moving in continuous space: Particles can be identified by their unique position in space, bimolecular reactions are triggered by collisions of particles, and typically particles diffuse by Brownian motion [11, 12].
4. Partial differential equations: Spatial gradients of concentrations are calculated deterministically [13].
5. Spatial stochastic dynamics: Multiple particles can occupy a position within a lattice space, and diffuse between neighboring positions [14].

Particle-based approaches (in the above categorization *c)*) are the subject of a further categorization suggested by SCHÖNEBERG et al. [8] (Fig. 1, right lower corner). Spatial particle-based simulation tools are categorized into those that support

- *level 1 – free diffusion*: basic 3D diffusion of point particles and reactions between them are considered,
- *level 2 – confined diffusion*: diffusion can be constrained to compartments,
- *level 3 – excluded volume*: point particles are replaced by volumetric entities, and
- *level 4 – potentials for particle-particle interaction*: instead of a simple rejection of movements assuming rigid cells, potentials are used to determine the interaction of particles in terms of exclusion of movement or reactions.

According to this categorization [8], Smoldyn is characterised as *level 2* particle-based approach: point particles which do not hamper each others movement are equipped with binding and unbinding radii, they can be constrained to compartments. A more recent version includes collision strategies to reach *level 3* [15]. SpringSaLaD [16], ReaDDy [17], and SRSim [18] are based on forces to model

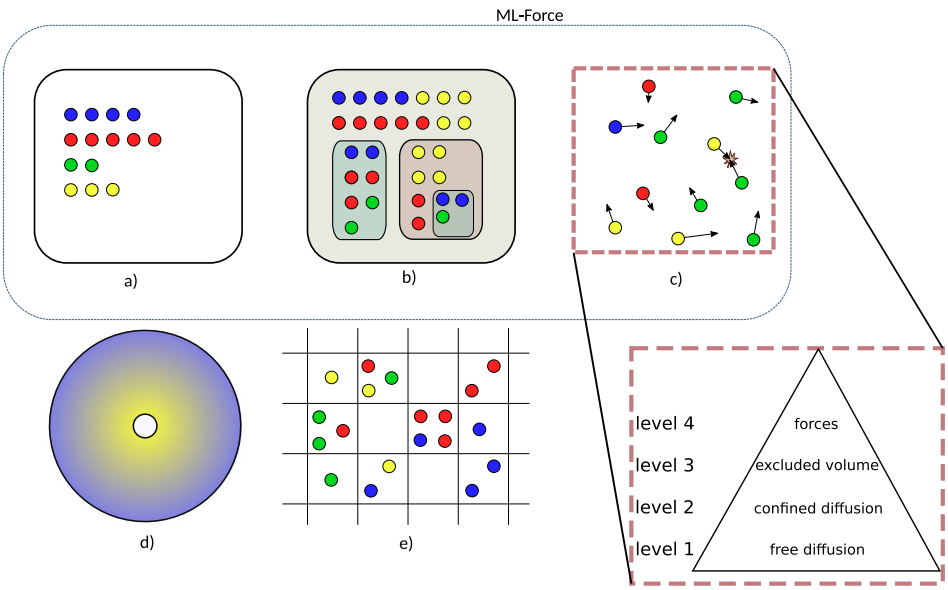


Fig. 1 Representations of space which is based on and slightly adapts Figure 1 of [7]. We distinguish between (**a**) stochastic non spatial dynamics, (**b**) compartmental dynamics, (**c**) individual particles moving in continuous space, (**d**) partial differential equations, and (*e*) spatial stochastic dynamics. Type (*c*) individual particles moving in continuous space is further described by four levels, according to SCHÖNEBERG et al. [8]. Each level builds upon the capabilities of the underlying ones, thus the representation as a pyramid. For example a method that provides excluded volumes should conceptually also be able to provide free diffusion and confined diffusion. ML-Force is characterized by a-b-c(4)

the particle-particle interactions (*level 4*). Still, the levels introduced do not imply that a tool that supports *level 3* provides all interesting features that a tool working at *level 2* offers to study the system of interest. E.g., DONOVAN [19], whose spatial simulation works at *level 2*, supports arbitrarily complex 3D mesh geometries, whereas in ML-Space [20] (*level 3*) only rigid spheres are considered.

Merging both characterizations, approaches that combine different spatial representations can be characterized. The Two-Regime method [21] and KLANN et al. [22] allow to combine particle and RDME dynamics within one model: *c(3)-e*. Similarly in [23], particles and partial differential equations are combined to focus on the spatial region of interest: *c(3)-d*. [24] couples a partial differential equations solver and Smoldyn *c(2)-d*. ML-Space combines particles at *level 3* (as excluded volumes are considered), compartmental dynamics, and spatial, stochastic simulation (RDME): *b-c(3)-e* [20].

In addition, simulation environments offer the possibility to select different spatial semantics for one model specification. For example, in VCELL [13], rule-based models can be interpreted by a particle-based simulator (i.e., Smoldyn) or a partial differential equation solver, both confined to realistically geometrical compartmental structures (*c(2)*, *d*). This list is far from being complete, and the characterization of the simulation tools may only depict a specific state in their development.

Against this background, we propose a new spatial particle-based modeling and simulation approach for cell biological systems, i.e., *ML-Force* (Fig 1). It combines:

- *Potential-based particle dynamics*: The excluded volumes that are represented by the large amount of macro-molecules within the cell affect the physico-chemical kinetics of various intracellular processes [25]. To capture the effects of molecular crowding, space exclusions need to be considered. In addition, interactions based on potentials are an effective means to study the formation of clusters [18]. Additionally, potentials allow us to capture directed movements, such as the transport of vesicles [26] (see "Results" section). Similar to ReaDDy [17], SRSim [18], or SpringSaLaD [16], ML-Force will use potentials for particle-particle interaction: *c(4)*.
- *Compartmental dynamics*: Intra-cellular space is further structured by compartments and vesicles, most of which are subject to frequent changes in terms of numbers, content, and inter-connectivity. A prominent example is the endosomal system. Vesicles form at the membrane. Here they acquire their cargo and engulf protein receptor complexes. Those are transported towards the inner cell, where part is degraded and part is recycled. The vesicles themselves move through stages of early sorting, recycling to late endosomes, closely interacting with each other – processes which include frequent fission and fusion [26] (Fig. 2). ML-Force uses the same force-based approach that it uses for the interaction of particles and global force functions, to model the compartmental dynamics. Similar to ML-Space, ML-Force supports both: compartmental and particle dynamics. However, unlike ML-Space it uses potentials *b-c(4)*.
- *Rule-based approach with arbitrary attributes and functions*: As other spatial simulators, ML-Force supports rule-based modeling [29]. However in ML-Force, attributes are not constrained to a finite set of values. Similarly, as in ML-Rules [30], attributes

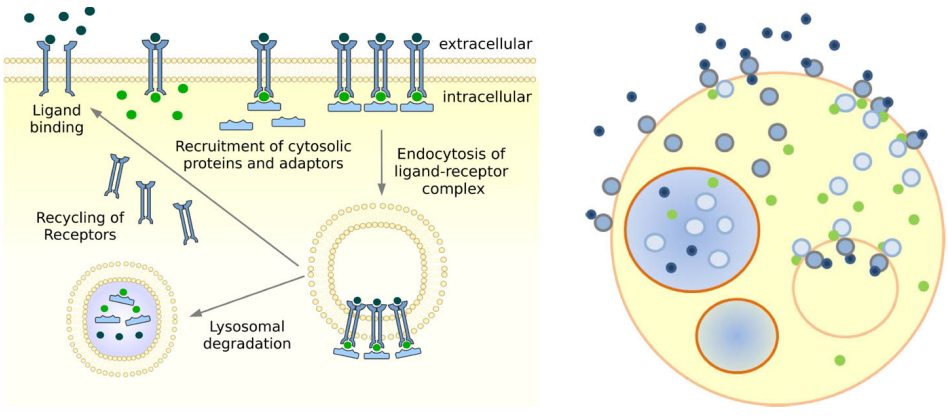


Fig. 2 Illustration (combined from [27, 28]) of a receptor-ligand binding kinetics, which includes receptor protein coupling, internalization, and recycling, whose components can be interpreted as nested arrangements of objects within objects that interact at intersection points, can be approximated to spherical particles of various sizes, move in continuous space and undergo processes of creation, degradation, internalization, externalization, fusion, and fission. **a** Biological view, **b** Hollow spheres view

are of arbitrary type and arbitrary functions are supported to determine the result and kinetics of reactions. The possibility to define attributed species and arbitrary functions increases significantly the expressiveness of a modeling formalism [31]. It allows to express spatial stochastic models (RDME) on a lattice (e) within a modeling approach that relies on a standard stochastic simulation algorithm for execution (a). By attributing species with indices that locate them in a specific point in the lattice (which represent a subvolume), e.g., $A(x,y), B(x,y)$, reactions can be constrained to species within one subvolume, e.g., $A(x_a,y_a) + B(x_b,y_b) \rightarrow 2\,B(x_b,y_b)@\,\text{if}\,x_a = x_b \wedge y_a = y_b\,\text{then}\,k\,\text{else}\,0$. Furthermore diffusion between neighboring subvolumes can be defined, e.g., $B(x_b,y_b) \rightarrow B\Big(\text{oneOf}\big((x{+}1,y),(h{-}1,y),(x,y{+}1),(x,y{-}1)\big)\Big)@k_d$. Thus, attributed species and arbitrary functions can be used to add space to otherwise non-spatial approaches. This is an observation also made in colored PETRI-Nets [32] and in expressive rule-based modeling languages such as ML-Rules [30].
In ML-Force, equipping particles with attributes and arbitrary functions allows e.g. to describe some part of the cellular dynamics in a non-spatial manner. For example, the reaction $B + C \rightarrow D\,@\,k$ which takes place in particle A can be transformed into a non-spatial stochastic first order reaction (a). Therefore, the particle A is equipped with the corresponding attributes and a reaction is defined $A(b,c) \rightarrow A(d)\,@\,b \cdot c \cdot k$. The function $b \cdot c \cdot k$ calculates the propensity of the reaction to take place. The possibility to constrain reactions based on arbitrary functions defined on the attributes, allows us to include non-spatial stochastic dynamics (a) which brings ML-Force up to *a-b-c(4)* (see Fig. 1).

Methods

ML-Force adopts a rule-based approach to specify the spatial biochemical dynamics. Firstly the mathematical concepts and abstractions are presented. This is followed by a short description of its implementation and embedded domain-specific modeling language.

Mathematical concepts and abstractions

As the name suggests, forces are central in ML-Force: they govern the interactions between and most of the activity of particles in a continuous manner. This is in contrast to the discrete event view chosen by simulators such as ML-Space or other agent-based approaches which interpret compartmental dynamics such as the proliferation of cells as discrete events [20, 33].

Representation of particles

ML-Force is a particle-based approach. Particles represent *hollow spheres*. As particles can contain other particles, they can be used to model cellular compartments. Particles are spherically symmetric and deformable, their size, properties and nested content may vary. Particles p_m are categorized into classes called *species*: $p_m \in S_i$. Species are characterized by a set of arbitrary attributes, $S(a_1, \ldots a_n)$, e.g., radius, phosphorylation sites, or phases. In our implementation of ML-Force we require particles to have at least the attributes radius r (non-zero) and density ρ. The information about its radius together with information about the system's temperature and the viscosity of its environment allows to derive the diffusion coefficient of a particle. In combination with the density, this information is used to determine the actual velocity (see "Spatial propagation" section). Particles may contain other particles, $p_m(v_1, \ldots v_n)[\,p_j + \ldots + p_k]$. All particles belonging to one class share the same attributes and attribute-types. Attribute values and content of particles may vary among particles of the same species. The interactions between and the activity of particles is governed by rules of behavior that operate on the particles, their attributes and their content. They rely on arbitrary functions for accessing and updating attributes and content of particles. Particles belonging to the same species share the same behavioral patterns.

Spatial propagation

For particles to interact they need to be in close proximity. In ML-Force, we base our model of propagation on the LANGEVIN equation [34]

$$m_i \ddot{\boldsymbol{x}}_i(t) = -\gamma \dot{\boldsymbol{x}}_i(t) + \boldsymbol{f}(t) + \boldsymbol{F}_{\text{ext}}(\boldsymbol{x}_i, t)\,, \tag{1}$$

with the dot denoting time derivative and boldness indicating vectors. A Langevin equation is an ordinary differential equation with a random term added. This equation can be intuitively understood as an extension to NEWTON's $\boldsymbol{F}(\boldsymbol{x},t) = m\ddot{\boldsymbol{x}}(t)$ by adding both friction (friction coefficient γ) which is linear to the velocity and a random force $\boldsymbol{f}$ which describes the accumulated effect of the system interaction with the individual particle. This interaction between the particles and the system results in the diffusion of the particles. In particular $\boldsymbol{f}$ is commonly sampled from a Gaussian distribution to model the aggregate system behavior.

Values for the constants involved can generally be found using known equations like STOKES-EINSTEIN relation ($D_i = \frac{k_B T}{\gamma_i} = \frac{k_B T}{6\pi\eta r_i}$), as well as system properties like viscosity η.

The $\boldsymbol{F}_{\text{ext}}(\boldsymbol{x},t)$ expression is the sum of all external forces on a particle. Two types of external forces are distinguished:

- global forces are those, that act on one particle, independent of other particles. Global forces functions may be specified as arbitrary functions of the individual particle state (most importantly its position) and the global state of the simulator. This may be used to model, for example, some global current in a blood stream or some other directed cellular motion (as is done in "A model of vesicular transport" section below).
- pair-wise forces are those, that originate in the interaction of two particles like the non-reaction force (Eq. 2), reaction force (Eq. 7) or a division force (Eq. 6).

Non-reactive force

In order to model excluded volumes, we introduce a soft sphere potential between two non-reacting particles. This is a common and natural way to model excluded volumes in force-based approaches. In addition, a hard-sphere potential would imply more efforts numerically, as it would lead to a discontinuous force. An established choice for the modeling of elastic (soft-sphere) collision is the Hertz collision theory [35]:

$$F_{non-react}(d) = \frac{4}{3}E^*\sqrt{r \cdot d^3} \tag{2}$$

where d is the overlap of the particles, $E^* = \left(\frac{1-\nu_1^2}{E_1} + \frac{1-\nu_2^2}{E_2}\right)^{-1}$ results from the Young's modulus E and Poisson's ratio ν and $r = \left(\frac{1}{r_1} + \frac{1}{r_2}\right)^{-1}$ from their radii. However, for many sparse applications, any kind of polynomial would suffice, as the exact dynamics of the elastic collision do not matter, when the mean time between collisions is sufficiently large.

Reactions in mL-Force

Reactions between particles in ML-Force include zero, first and second order reactions. ML-Force supports a rule-based description of models. Whereas lower order reactions are sampled from an exponential distribution, second order reactions are calculated, similarly as the movement of particles, based on forces.

Lower order reactions

Simulation of zeroth and first order reaction, e.g., $\perp \rightarrow p_j$, or $p_i \rightarrow p_i'$, or $p_i \rightarrow p_j + p_k$ is a well understood problem (e.g., in SpringSaLaT [16]). The most basic approach is to sample the rate every (sufficiently small) time-step and check it against a random number.

In ML-Force, rates can be defined by arbitrary functions that access the attributes and contents of the particles involved in a reaction, e.g., for a first order reaction:

$$p_i(\nu_1, \ldots \nu_n)[p_1 + \ldots + p_m] \rightarrow \ldots @\lambda(k, \nu_1, \ldots \nu_n, [p_1, \ldots, p_m])\ . \tag{3}$$

here λ denotes a function defined by the user, k a parameter, which might refer to the system globally, e.g., in terms of temperature and viscosity, or to a kinetic constant. The result of the function λ serves as the parameter for the exponential random distribution, that describes the probability density of the time until the reaction occurs. Thus in ML-Force, zero and first order reactions are treated semantically the same as in stochastic simulation algorithms [9]. In stochastic, non-spatial and spatial, simulation approaches (see Fig. 1a and e) exponentially distributed stochastic events form the basis of all, including second order, reactions.

In ML-Force the rate is sampled every time step. Instead of propagating the simulation until the exact time point a reaction occurs, we check (stochastically), if the reaction occurs in the current time step. Sampling the rate every time step is useful if the information $(\nu_1, \ldots, \nu_n, [p_1, \ldots, p_m])$, on which the rates depend, frequently changes. If this information remains largely constant over time, a scheduling approach may be more efficient computationally, as events do not have to be rescheduled frequently [36]. In executing ML-Force models so far, zeroth and first order reactions have only been responsible for a small fraction of the required computing-time.

In addition to computational considerations, the error introduced by this procedure needs to be considered. Due to the discretization of time (explicit time steps) no-more than one reaction per particle may take place per time step. This is important to keep in mind for future models. In the application domain the mean time between reactions tend to be large compared to the very small time steps that result from the fast diffusion of particles. As long as this is the case, the error is negligible. With resonable technical effort, this problem could also be avoided by allowing multiple reactions per time step.

One kind of lower order reaction is the changing of particle size, in terms of a particle's radius r. In ML-Force arbitrary functions can be applied. The new value of the radius r, i.e., ν_1', of a particle $p_i \in S_j(r, \ldots)$ is calculated in dependence of some global parameters k, and the current state of the particle in terms of its attribute values $\nu_1, \ldots \nu_n$, and its content $p_1 \ldots p_m$, such that

$$p_i(\nu_1, \ldots \nu_n)[p_1 + \ldots + p_m] \rightarrow p_i(\lambda(k, \nu_1, \ldots \nu_n, [p_1, \ldots, p_m]), \nu_2 \ldots, \nu_n)[p_1 + \ldots + p_m] \ldots @\lambda(\ldots)\ .$$

By default, whenever a particle is created, it is first put into the system as a very small particle and then quickly grows continuously to its desired size. This continuous change of particle radii is also used, whenever the radius

of a particle is changed in the model by any kind of reaction. Carrying the changes out suddenly (so not slowly over multiple integration steps) would potentially lead to integration errors, especially in systems with nesting. A sudden change in size could for example lead to a high particle overlap, which in turn would create a high unrealistic potential energy leading to a strong force and thus a disturbance in the system, e.g., numerical heating. Slow changes in size, allow for each artificial change in energy to slowly dissipate e.g., in friction due to the numerically very stable Langevin model. Naturally, it is also possible to create a particle within or at the surface of another particle. Those new positions are chosen randomly. At the surface this means that the new particle's shell barely touches the old particle's shell.

Compartment division

An interesting kind of first order reaction is that of a cellular compartment (or specifically a cell) dividing.

$$p_i(\nu_1,\dots\nu_n)[p_1+\dots+p_m] \to p_{i1}\big(\lambda_{11}(y),\dots,\lambda_{n1}(y)\big)\big[\lambda_{c1}(y)\big] + p_{i2}\big(\lambda_{12}(y),\dots,\lambda_{n2}(y)\big)\big[\lambda_{c2}(y)\big] \qquad @\lambda(\dots),\{a\}\,, \tag{4}$$

with $y = k, \nu_1, \dots \nu_n, [p_1, \dots, p_m]$ or in shorthand, omitting attributes and contained particles

$$p_i \to p_{i1} + p_{i2}@\lambda(\dots),\{a\}\ . \tag{5}$$

In ML-Force this is realized in a continuous and nesting compatible manner. As a first order reaction, the event that the division starts is sampled from an exponential distribution. The simulator temporarily introduces an "ignoring" relation between particles. This can be best explained using an example: Imagine a particle p_i containing many particles $\hat{p}_1 \dots \hat{p}_m$. If now the reaction for p_i to split into p_{i1} and p_{i2} triggers, a few things happen. First of all, each particle $\hat{p}_1 \dots \hat{p}_m$ gets assigned to remain in either p_{i1} or p_{i2}. By default, this assignment happens randomly. If required by the model, a specific splitting function may be specified by the user. Therefore, two arbitrary functions, i.e., $\lambda_{c1}, \lambda_{c2}$ are applied. The results of these applications form the new content of the newly generated particles, i.e., p_{i1} and p_{i2} respectively. The functions λ_{i1} to λ_{n1}, respectively λ_{i2} to λ_{n2} determine the new attribute values of the particles. If attribute values remain the same, simply the old values ν_i can be used.

Each $\hat{p}_i$ that will remain in p_{i1} is set to ignore the particle p_{i2} and vice versa. Also p_{i1} and p_{i2} are set to ignore one another. Now an external force is applied to push p_{i1} and p_{i2} apart

$$F_{\text{divison}} = m_i \cdot a_{\text{division}}\ . \tag{6}$$

The force is calculated based on the acceleration a that is part of this particular reaction type (Eq.: 4, 5) in ML-Force and the mass of the involved particles (which again depend on the respective radii and their density ρ). This is chosen, instead of a uniform force, to create a uniform motion of all particles. In principle however, any force expression suitable to the model could be used here, like a half sine wave for example. Once p_{i1} and p_{i2} are fully outside one another, all the newly introduced ignore relations are removed and regular propagation proceeds.

The principle of continuous change in particle size also applies here. At the beginning both particles have the same size. If p_{i1} and p_{i2} are to change their size throughout the division (see example in "The yeast model" section), this change is carried out throughout the division process. Initially, both p_{i1} and p_{i2} share position and radius. The further they move out, the closer their radii become to the desired ones. The fusion of compartments could be added in an identical (if reversed) fashion. However so far, it has not been implemented yet.

Bimolecular reactions

Handling bimolecular reactions in particle-based approaches is rather challenging. Whilst solutions exist when not considering excluded volumes (e.g. Smoldyn [12]), there has not yet been agreement among the community on the best way to approach this problem. This is evident from the many different approaches of recent tools [5, 12, 16, 17, 37]. Two processes contribute to the macroscopic rate,

- the *diffusion process* describes how likely it is that two particle actually move into the vicinity of each other. This upper bound to the reaction rate is well described by the SMOLUCHOWSKI-Theory [38] (elaborated in the Additional file 1)
- the *microscopic activation process* describes how likely it is for two particle to react once they are close to each other [39, 40].

The rate is considered to be diffusion limited if every interaction results in a reaction, all other cases are considered to be activation limited. We propose to use forces for both processes. This means also the microscopic activation process is based on forces. In case a reaction can occur between two particles, the non-reactive force (Eq.: 2) has to be replaced by a reactive force. We used a rather simple approach which is calculated based on a degree of overlapping required d^* and the work to cross the energy barrier between the two particles. Both, the degree of overlapping and the energy barrier, are specific for the reaction to occur and have to be defined by the modeler.

$$F_{react} = \frac{W}{d^*} \tag{7}$$

where d^* denotes the required overlap of the particles and W the work required to cross the energy barrier. It should be noted due to the modular design (see "Simulator and implementation" section) within the simulator these functions can easily be changed. Suppose we were to look at the reaction $p_i + p_j \rightarrow p_k\{d^*, W\}$. Whenever p_i and p_j overlap, they start to repel each other with the force F_{react}. Once they overlap more than d^*, we consider the reaction to be triggered. Thus, the energy barrier and the required overlap of the microscopic interaction potential determines the likelihood of the reaction to take place. The higher the energy barrier, the less likely the particles will overlap sufficiently. If the energy barrier W is negligible, the diffusion is the only limit to the macroscopic reaction rate in the reaction process, it is *diffusion limited*.

This type of bimolecular reaction handling with forces, allows also for self-consistent handling of dynamic nesting (and "unnesting") reactions, i.e., one particle shuttles into (or out of) another particle. In principle the same concepts apply as for the regular bimolecular reaction. Looking at the reaction $p_i + p_j \rightarrow p_i[p_j]\,\{d^*, W\}$ (and the same for $p_i[p_j] \rightarrow p_i + p_j\{d^*, W\}$), first the particles need to diffuse into proximity of one another. Once particles overlap, they repel each other. If p_i's diameter has fully passed over p_j's outline, the nesting reaction is triggered. However at this point p_j is already fully contained in p_i so there is no jerkiness in the simulation. For example, if there is no unnesting reaction defined for these particles, the only thing, that changes, is that, if p_j were to move towards again touching p_i, it would perceive the force $F_{non-react}$ (pulling inwards). This ensures a soft-sphere-style excluded volume. Similar as for the regular $p_i + p_j \rightarrow p_k$ type bimolecular reaction, a high energy barrier makes for a less, and a low for a more likely reaction.

The advantage of this force-based approach, is the smooth integration of the nesting process. On the other hand, it does require a very small timestep, compared to other methods. Furthermore, in the current situation, the parameterization of bimolecular reactions is not optimal. Ideally, the modeler would specify the macroscopic rate, instead of the microscopic barrier. In the future we plan on investigating this further and, using principles from thermodynamics, provide an automatic conversion method. This would mean, by modeling the system as a thermodynamic ensemble we could determine the statistical likelihood of a particle surpassing a certain energy threshold during interaction and put this into explicit mathematical relation to the reaction rate.

Simulator and implementation

The simulator has been implemented using `C++14` and has been tested on both Windows and Linux (using CMake-build system) and is available at https://git.informatik.uni-rostock.de/mosi/ml-force-publication.

The user-interface is built around a rudimentary embedded domain specific language (DSL). An example and explanation can be found in the provided code snippet below. In the long run, it would be advantageous to create an external DSL using techniques like transpiling, to still have the performance benefits of a compiling language. A more in-depth comparison and discussion of DSLs can be found in the literature [41]. The developed embedded DSL bears some similarities with the ℓ-language [42] which also supports an imperative rather than a declarative approach towards modeling.

As can be seen in Fig. 3, the software pursues a component-based design with a clear separation of concern [43]. Its components are arranged around the main *simulation engine*. The simulation engine itself is very basic and designed to make as little assumptions about the underlying semantics as possible. This simulation engine interacts with the components via a well defined, small interface. Each of the key ingredients of the simulation is separated into exchangeable components, namely the integrator, collision detection engine, and the visualizer, as well as the overarching modeling layer which provides the interface between the simulation engine and DSL. They are invoked by the simulator and implement the functionality introduced in the previous section. The simulator is responsible for executing the lower and higher order reactions by calling the specified functions of the model. First it is checked whether lower order reactions trigger in this step. If so, those are executed. Based on the forces a numerical integrator calculates the new positions of the particles. To identify the particles that interact, a collision detection engines is invoked. Afterward the simulator calculates the bimolecular forces and executes the biomolecular reactions accordingly. The calculated state can be subject to visualizations or other reporting mechanisms.

Integrator Currently integration is carried out using a two-step kick-drift-kick integrator [44] of the stochastic Langevin equation. This integration includes velocity, and is thus fairly costly numerically. However as an alternative, one could select an integrator with immediate dissipation of momentum, that would allow for a considerably larger timestep. The choice for the more complex integrator was made as ML-Force makes heavy use of various forces and carrying out the more complex integration, helps in terms of correctness. In the current implementation the time step is chosen adaptively based on the smallest particle and the fastest velocity currently in the system, as $\Delta t = \frac{r_{\min}}{v_{\max} g}$ with g as the parameter of *granularity*, that describes the precision of the integration.

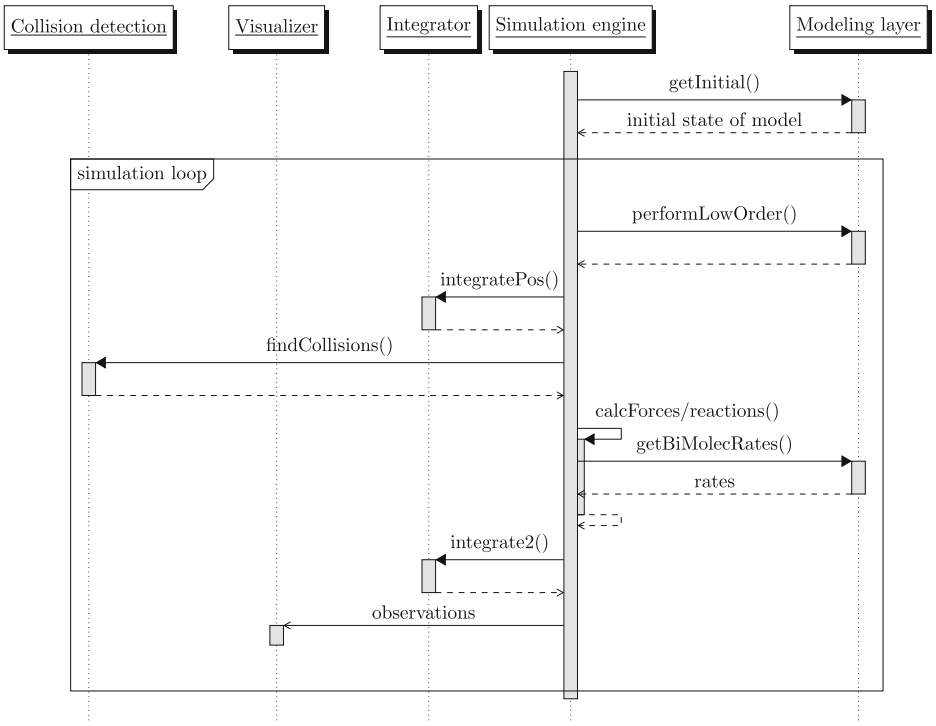


Fig. 3 The different components of the software and their respective tasks, as called upon by the main simulator in an UML sequence diagram. The different components of the software are interchangeable and only interact via their interface to the main *Simulator*-component. This simulator also provides the layer between the model (as specified using the DSL).The software implementation is described on in "Simulator and implementation" section. The second integration of the particles is optional. For example the current integrator advances the velocity in the second integration step

Collision detection engine Finding possible interactions of particles using the collision detection engine is one of the key drivers of runtime in the current implementation. Several collision detection engines have been implemented, using different algorithms and parallelization. For example, for highly nested particles, the collision-detection problem lends itself to be solved by a specifically tailored algorithm and a massively parallel execution on the GPU [28].

Visualizer Currently 3 basic visualizers have been created. One to just create an output text file, and two based on CImg [45], one for live viewing and one for generating movie files. The visualizers are encapsulated in a separate thread to minimize overhead. One more advanced visualization has been implemented that maps the simulation to 3D space using OpenGL [46]. It offers some useful features for the case studies (see "Results" section), such as tracking individual particles in space.

Modeling layer The modeling layer makes heavy use of λ-functions in order to encapsulate model specific behavior into single function calls for the simulator. λ-functions here are a `C++11` feature that allows to define anonymous functions, that can be used as parameters in a functional-like programming style. This is facilitated by various templates that have been defined in the DSL. For example, after each reaction, the simulator invokes the *post reaction function* (see e.g. afterFuncA in listing 2). In the modeling layer this function is specified by the user and denotes the results of applying a rule, such as particles changing state, being degraded or created.

This simplifies the implementation of the simulator and gives a unified and lean interface for implementing further features. The simulator also provides a set of units (via the `units::` namespace) to the modeler, allowing for consistent parameterization. For simplicity the reactants in the DSL are always denoted as *A* and *B*. The particular type of *A* and *B* is then specified via the *as_A* etc. functions. In general the DSL is very expressive, as all rates and reactions are described as λ-functions. These can be of arbitrary complexity. However, it should be noted that the current DSL presents only a proof of concept rather than a language that allows a succinct description of ML-Force models. Therefore, further efforts will be dedicated to improving the design of the language.

Listing 1 This is an extract from the lipid raft model that describes how a receptor moves into a lipid raft. The embedded domain specific language is built around objects. *Species*, *reaction* or *property* objects can be put in relation to one another. Some macro-functions are made available, to minimize boilerplate code (e.g. the `ADD_SPECIES`-macro does not only create a species object, but also sets a string property with that object's name). The `*_FUNC` are wrappers for different λ-functions that can be used to specify arbitrary complex behavior and rates.

```
/* declaring species and properties */
 ADD_SPECIES(lipid_raft);
 radius(lipid_raft) = 200 * units::nano_meter;
local_density(lipid_raft) = nature::protein_density
    ;
local_viscosity(lipid_raft) = 10E-3 * units::pascal
     * units::second;
colorType(lipid_raft) = 8;

ADD_SPECIES(protein);
local_density(protein) = nature::protein_density;
radius(protein) = 3.7 * units::nano_meter;
colorType(protein) = 1;

/* declaring A + B -> B[A] reaction */
reaction Protein_in(as_{A}(protein), as_{B}(
    lipid_raft));
Protein_in.barrier = 0. * units::kilogram *
               pow(units::nano_meter/units::second
                   ,2);
Protein_in.AinB();

/* inital state */
putHere(lipid_raft, box_size / 2, box_size / 2, 0);
putSomewhere(protein, 200);
```

Results

In order to test the ML-Force simulator and its implementation, we conduct a few case studies. These include simple test cases to investigate correctness and more complex models. The latter are inspired by realistic cell biological models, make use of the features in ML-Force, and, thus, demonstrate their usefulness. Both, the basic test models and the more realistic ones, can be found in the Additional file 1.

Testing correctness

As other particle-based simulation approaches [16, 17, 37, 47] we test the correctness of our simulator, based on a set of simple reactions which we simulate to compare the achieved results to theory. To those tests belong the creation of a particle with a constant rate as a 0 order reaction ($\emptyset \xrightarrow{k} A$), the decay with a rate $A \xrightarrow{k} \emptyset$ as first order reaction, and irreversible and reversible second order reactions $A + B \leftrightarrow C$ in the diffusion limited (intrinsic rate $k_{int} = \infty$) case. A comparison of simulation results and theory shows overall a good agreement with the theory (see Additional file 1). However, those tests also reveal the decisive role of selected parameters to determine forces ("Bimolecular reactions" section) and time steps. For their selection, suitable computational support should be provided. As a form of pre-processing step, energy barriers respectively accelerations for individual reactions could be generated by automatically fitting simulation results to theory. A similar strategy can help selecting suitable time steps, so that the diffusion of particles is correctly simulated (see Additional file 1).

A model of vesicular transport

Heinrich and Rapoport proposed a model of vesicular transport [48]. Although the model describes a spatial process, it was formalized by means of differential equations.

The vesicular transport model refers to two cellular compartments which exchange membrane-bound soluble N-ethyl-maleimide–sensitive factor attachment protein receptors (SNAREs) and cargo proteins with the help of differently coated vesicles. These are budding from the compartments and move, driven by motor-proteins, to the other compartment where they are fused. Thereby the coat of vesicles defines, which type of SNAREs (X or Y), cargo proteins and motor-proteins are bound to them. Since different types of motor-proteins move in different direction, SNARE X and Y accumulate in different compartments.

By modeling this process in a spatial regime, the problem starts, as in each particle-based approach, with the description of the species. Each species needs a radius and a diffusion coefficient. The ML-Force simulator determines the diffusion based on the temperature, solvent viscosity and particle radius according to the Stokes-Einstein equation (see Additional file 1). The cell, compartments and the vesicles are represented as particles while the SNAREs are attributes of particles, which change during the budding and fusion processes. As Klann et al. [2] has already shown through the translation of the original ODE model into a spatial agent-based model, this transport relies on a directed movement of vesicles. Therefore Klann et al. added a cytoskeleton structure to the cell on which the motor-proteins moved along. In ML-Force the direction is determined by forces which take the type of particles and their attributes into account. An external force field is introduced which is shaped like a dipole field with the compartments as poles. Based on the coat of the vesicles a force along this field is added to their movement, which lead them to one of the compartments. To test the sorting mechanism of this simple vesicular transport model we study a system with 2 types of SNAREs (X and Y) with the same amount. As in [48], we initialize the model: the first compartment is nine times bigger ($V_1 \approx 0.118\ \mu m^3$) than the second one ($V_2 \approx 0.013\ \mu m^3$) and contains 90% of the SNAREs (X and Y) ($N_1^{X/Y}$ = 90000 and $N_2^{X/Y}$ = 10000). Since the budding process of the vesicles depends on the volume of the compartments, they should reach an equal size and the SNAREs should be sorted.

As shown in Fig. 4 both compartments reach an equal volume and SNARE X accumulates in compartment 1 while SNARE Y accumulates in compartment 2 (not shown).

To show that the directed motion is essential for the sorting, as a control we run the simulation without the external force field and with compartments with equal initial size ($V_{1/2} \approx 0.065\ \mu m^3$) and equally initially distributed SNAREs ($N_{1/2}^{X/Y} = 50000$). In a purely Brownian model, i.e., a model where all particles perform Brownian motion, no sorting takes place and both compartments are just shrinking as shown in Fig. 5. It can also be seen that the vesicle's slowly diffusion around the compartments and there behavior is independent of their coating. In contrast the vesicles in the model with a directed movement either re-fuse with the compartment of origin or move to the other compartment based on their coating. As a result vesicles in this model fuse with a specific compartment after a short time and by this they are sorting the SNAREs. This simplified model illustrates how external forces can simulate the directed motion in a biological system.

The yeast model

A more complex model that showcases more of ML-Force's abilities is based on an illustrative yeast model [30], which in turn uses the model of the cell cycle proposed by TYSON [49]. In the original model each yeast cell can grow and contains five different proteins whose amounts oscillate periodically and trigger the fission of a yeast cell. Each cell has a mating type (P or M) which can change during the cell fission. Depending on the type the yeast releases pheromones (M-factor and P-factor, respectively) to inhibit the cell cycle of cells with opposite mating type. In addition, the type M cells secrete a protease called Sxa2 which inactivates the P-factor pheromone that stems from the type P cells.

With the ML-Force model, we want to check whether cells of identical mating types accumulate in some areas and inhibit the cell cycle of cells with opposite mating type in this area. In ML-Force, each species of the system can be modeled as a particle, as shown in the upper part of Fig. 6. The problem of this approach are the different time scales of the dynamics. The cell cycle lasts about 120 min, while the diffusion of the proteins needs a time step in the sub nanosecond regime. In addition, for our question, a spatial simulation of the intra-cellular dynamics is not required. Therefore, we represent the pheromones and the cells as particles and the proteins involved in the cell cycle (as well as the cell cycle) as attributes of the cells, as shown in the lower part of Fig. 6. Whereas cells only move during the fission process in the x-y plane, pheromones and Sxa2 diffuse in the extracellular medium and disappear when they reach the borders of the test volume.

As shown in Fig. 7 after 900 min there are twice as many P-type cells as M-type cells. Furthermore some of the M-type cells are old (large) and have a highly inhibited cell cycle, which makes it likely that these cell will enter apoptosis before they can divide.
Attributed particles and arbitrary functions allow to model the intra-cellular dynamics non-spatially as first order reaction.

Listing 2 The creation of cyclin inside the cell is realized as a first order reaction that changes the attribute Y.

```
// 1) cyclin synthesis in cell 0 -> Y
reaction Y_synthesis(as_{A}(cell));
Y_synthesis.rateFunc = R_FUNC(k1);
Y_synthesis.afterFuncA = G_FUNC(Y(A) = Y(A) + 1;);
```

Nevertheless these non-spatial dynamic can influence the particle, here by triggering the division of a cell.

Listing 3 After the attribute 'Ma' of the cell drops below the threshold t9 the division reaction is executed and new cell namely B is created.

```
9) cell division (transition from M->G1)
reaction division(as_{A}(cell));
division.reacPossibleFunc = B_FUNC(Ma(A) < t9 phase
    (A) == 3);// #M_A < t9
division.rate = k9;
division.divide(as_{B}(cell), 1.E-8);
division.afterFuncA =
G_FUNC(volume(B) = 0.5 * volume(A); volume(A) = 0.5
    * volume(A););
```

It is also possible for a particle to influence the attributes (non-spatial) of particle.

Listing 4 The collision of the two particles cell and F_M results in the 'death' of F_M and the change of the attribute 'receptor_occupied' of the particle cell.

```
/* cell(n) + F_M -> cell(n+1) */
reaction bind_pherome_{M}(as_{A}(cell), as_{B}(F_M)
    );
bind_pherome_M.barrier = 0.;
bind_pherome_M.setOverlapp(0.01);
bind_pherome_M.reacPossibleFunc =
    B_FUNC(receptor_occupied(A) < receptor_total(A)
        && mating(A) == 1);
bind_pherome_M.afterFuncA =
    G_FUNC(receptor_occupied(A) = receptor_occupied
        (A) + 1;);
bind_pherome_M.Bdead();
```

For example this is the case in the yeast model when a pheromone reacts with a cell (both particles) and inhibits the cell cycle of the cell. This combination of spatial and non-spatial behaviors enables the simulation of systems which include processes on different temporal scales.

Lipid rafts model

The role of lipid rafts in inducing and promoting receptor accumulation within the cell membrane and the recruitment and binding of proteins from the cytosol has been

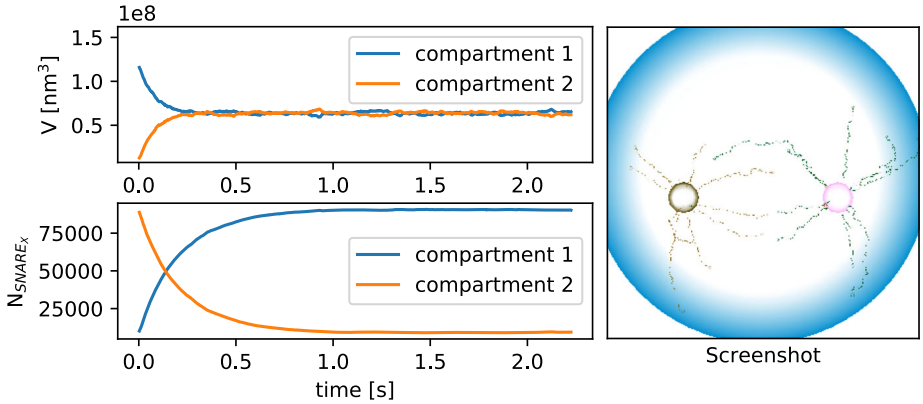


Fig. 4 Volume and amount of $SNARE_X$ over time and Snapshot from the vesicle transport model. In the case of a directed movement the volume of the compartments become equal and $SNARE_X$ accumulate in compartment 1 as shown on the left. The right side shows a screenshot from the simulation where you can see the movement along the external force field. The blue sphere is the cell with a radius of 2.5 μm

subject to several computational studies [50, 51]. A small model of receptor lipid raft interaction which is inspired by a sub-model of Wnt signaling [52] shall illustrate the ability of ML-Force to support dynamic nesting. The model consists of two proteins (LRP 5/6 and CK1-γ) which diffuse in the membrane and may enter or leave lipid rafts. In the model, the mobility of proteins is reduced in lipid rafts by a 10 times larger viscosity in the rafts compared to the rest of the membrane. Both proteins have different affinities to enter the lipid raft which is modeled by different energy barriers in the nesting reactions. LRP 5/6 has to pass a barrier of $12 \cdot 10^{-21}$ J which was estimated from the distribution of the kinetic energy for demonstration purposes. It shows how the energy barrier at a bimolecular reaction can be used to slow down the reaction. CK1-γ can enter the lipid raft freely ($E_{barrier}$ = 0 J). By choosing a larger energy barrier for LRP 5/6 while they are entering the lipid raft we model their lower affinity to the lipid raft and can show how the energy barrier at a bimolecular reaction can be used to slow down the reaction. To show how these functions influence the accumulation of the proteins in the lipid raft we run a 2D simulation. In this simple experiment, our model contains a single lipid raft which covers 25% of the surface. The 10 times larger viscosity of the lipid raft causes a 10 times lower diffusion coefficient in the raft compared to the rest of the membrane. The simulation starts with 200 LRP 5/6 and 200 CK1-γ particles outside the lipid raft. To enter the lipid raft, the particles need to overcome the energy barriers described above. After a particle entered a lipid raft they can leave it again without any energy barrier.

As expected, Fig. 8 shows that due to the slower diffusion in the lipid raft CK1-γ accumulates in the lipid raft.

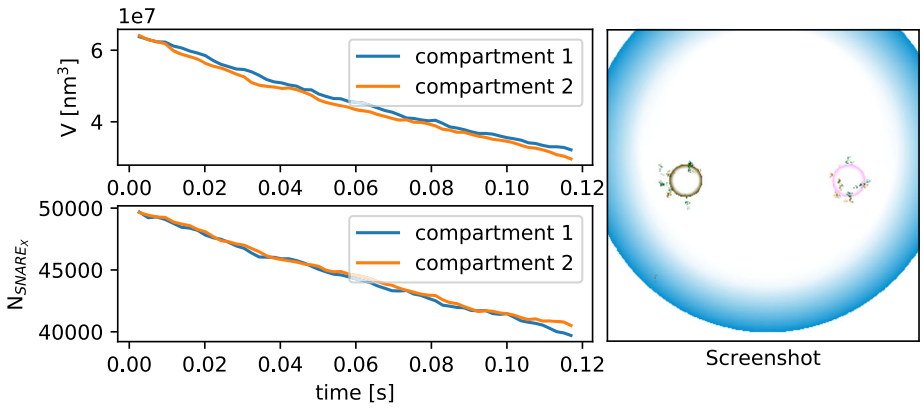


Fig. 5 Volume and amount of $SNARE_X$ over time and snapshot from the vesicle transport model. In the case of pure brownian motion the volume of the compartments decreases and no sorting of SNAREs takes place as shown on the left. The right side shows a screenshot from the simulation where you can see the random movement of the vesicles. The blue sphere is the cell with a radius of 2.5 μm

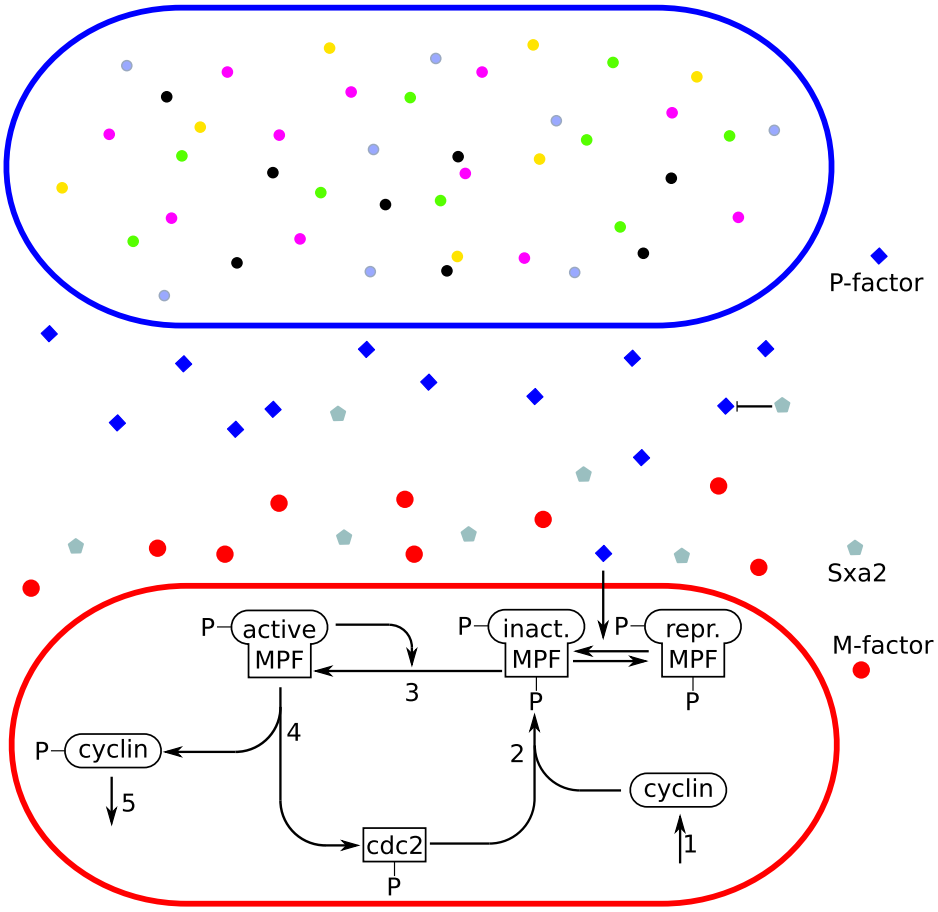


Fig. 6 Sketch of the spatial/non-spatial cell cycle. Cell cycle of a yeast cell with five/six different proteins. The upper cell shows the cell cycle modeled with particles. As the proteins interaction within the cell occurs on a vastly different timescale than the behavior of the cell this is not a feasible approach. Instead the internal dynamics are modeled by a non-spatial approach as shown at the bottom

The amount of LRP 5/6 which accumulates in the lipid raft is lower due to the energy barrier which hampers receptors to enter the lipid raft.
The model shows how the repulsive force during a bimolecular reaction can lower the reaction rate. Also the dynamic nesting of particles can change their behavior (here the diffusion of the nested particle) during simulation. It is also possible to restrict reactions to only occur if particles are nested inside a specific type of particle, such as the phosphorylation of LRP 5/6 which is constrained to lipid rafts.

Discussion

The benchmark models which range from the multicellular yeast model to the subcellular lipid raft model have shown the usefulness of the realized features. ML-Force expands upon the state of the art. The combination of compartmental dynamics and particle simulation based on forces is a unique feature of ML-Force. This feature has been used in the lipid raft model ("Lipid rafts model" section). ML-Force provides a consistent force-based semantics for spatially resolved dynamic nesting with excluded volumes. Arbitrary functions and attributes have proven a necessity for a concise and computationally feasible realization of the yeast model in ML-Force ("The yeast model" section) where they are used for the non-spatial stochastic simulation of the cell cycle. In ML-Force, intra-particle dynamics can be simulated either spatially resolved by nested particles or in a non-spatial stochastic manner. Applying lower and higher spatial resolutions on demand facilitates modeling and simulating complex spatial models. The vesicle transport model ("A model of vesicular transport" section) relied on the possibility to let particles grow and to define global force functions that apply to all particles independently of other particles to simulate the directed movement of vesicles. The global force functions provide an additional means for the modeler for abstraction (in this case from the cytoskeleton

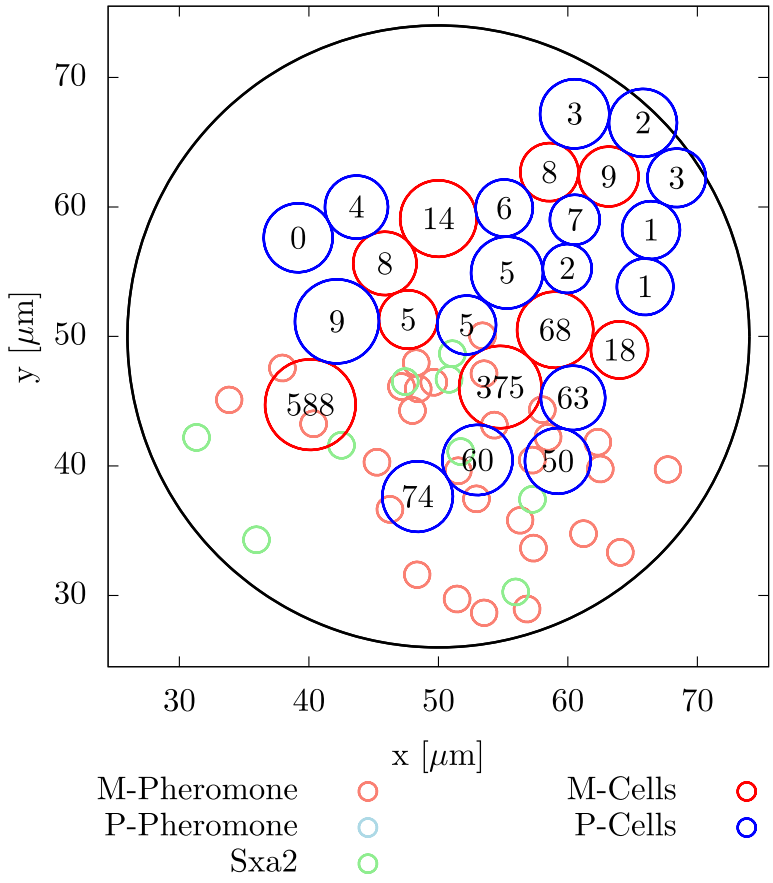


Fig. 7 Spread of yeast cells. Distribution of yeast cells after 900 min. The numbers in the cells indicate how many pheromones are bound to them. For example the M-type cell on the left site has a highly inhibited cell cycle with 568 bounded pheromones. It is very likely that this cell will enter apoptosis, before it divides

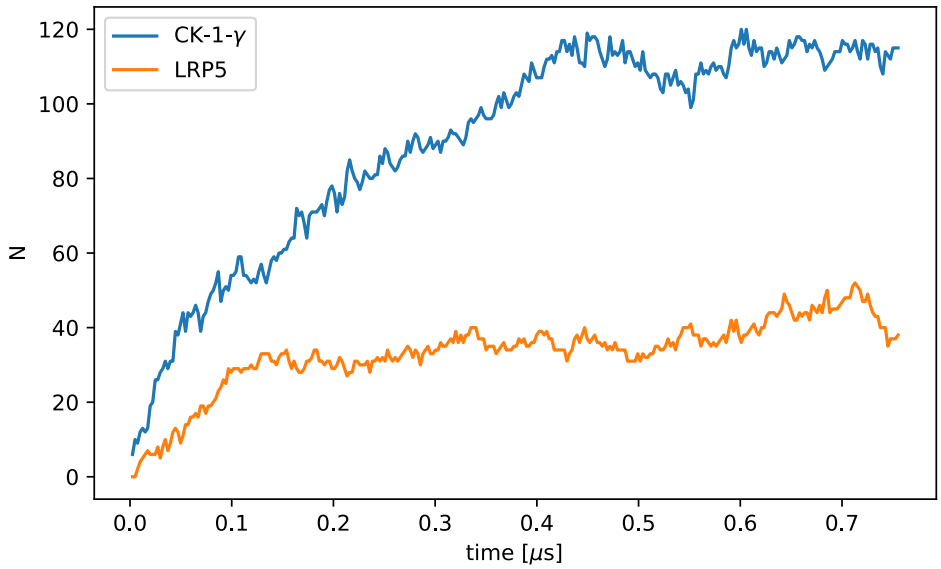


Fig. 8 Accumulation of proteins in lipid rafts. Amount of LPR 5/6 and CK1-γ inside the lipid raft. Due to the lower affinity of LRP 5/6 to the lipid raft (realized by an energy barrier) their concentration is lower then the one of CK1-γ

structure on which the motor-proteins move). Along all other dynamics this is seamlessly integrated into the force-based semantics of ML-Force.

Conclusions

The particle-based modeling and simulation approach ML-Force combines excluded volumes and forces with support for dynamic nesting (compartmental dynamics) and the ability of constraining cellular dynamics by arbitrary attributes and functions. Whereas other simulation approaches have treated compartmental dynamics, such as particles entering or leaving a compartment, or compartmental fission or fusion, discretely, in ML-Force those are simulated in a smooth, continuous manner. Thereby, structural dynamics are seamlessly integrated into the particle-based simulation governed by the LANGEVIN equation. However, this adds to the requirements the integrator has to face and consequently the induced calculation efforts.

ML-Force utilizes a rule-based modeling approach. In combination with attributed species of arbitrary types and arbitrary functions that work on particles, their attributes and content, this contributes to the expressiveness and flexibility of ML-Force. In particular, part of a spatial model can be executed in a non-spatial stochastic manner. Abstracting from spatial details on demand reduces the complexity of the model and the induced calculation effort and allows applying ML-Force to a wider range of cell biological systems. The possibility to define force functions that act on all particles independently of others provides another valuable means for abstraction in ML-Force.

Currently, ML-Force is being used in a combined in-vitro and in-silico study to analyze the impact of external electrical fields on cellular membranes. Future work will be aimed at enhancing the readability of ML-Force models by a more declarative expression of rules. In addition, user support for selecting time steps and suitable parameters for force calculation needs to be provided. Also advanced numerical integration schemes that are exploited by other particle-based simulators and their impact on compartmental dynamics shall be analyzed.

Supplementary information

Supplementary information accompanies this paper at https://doi.org/10.1186/s12859-019-3092-y.

Additional file 1: The additional file contains tests of of basic functions and the code of the models from the case study.

Abbreviations

CK1-γ: casein kinase 1-γ; DSL: domain specific language; LRP5/6: low-density lipoprotein receptor-related protein 5/6; ML: multi-level; RDME: reaction diffusion master equation; SNARE: soluble N-ethyl-maleimide–sensitive factor attachment protein receptors;

Acknowledgements

Felix Hauptmann implemented the OpenGL visualizer.

Authors' contributions

TK conceived the method. TK designed and implemented the Software. PH carried out the Case study. TK, PH and AMU wrote the manuscript. AMU supervised the project. All authors have read and approved the manuscript.

Funding

Financial support (design of the study, analysis, and interpretation of simulations and in writing the manuscript) was provided by the Deutsche Forschungsgemeinschaft (DFG) research grant 'ESCeMMo' (UH-66/13) and the DFG collaborative research centre 'ELAINE' (CRC 1270). We acknowledge financial support by Deutsche Forschungsgemeinschaft (DFG) and Universität Rostock within the funding programme Open Access Publishing.

Availability of data and materials

The datasets used and/or analysed during the current study are available from the corresponding author on reasonable request.

Ethics approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Received: 14 March 2019 Accepted: 10 September 2019
Published online: 27 November 2019

References

1. Scott J. D., Pawson T. Cell Signaling in Space and Time: Where Proteins Come Together and When They're Apart. Science. 2009;326(5957): 1220–4. https://doi.org/10.1126/science.1175668 Accessed 23 July 2019.
2. Klann M, Koeppl H. Spatial Simulations in Systems Biology: From Molecules to Cells. Int J Mol Sci. 2012;13(6):7798–827. https://doi.org/10.3390/ijms13067798.
3. Bittig AT, Uhrmacher AM. Spatial modeling in cell biology at multiple levels. In: Proceedings of the 2010 Winter Simulation Conference. Baltimore; 2010. p. 608–19. https://doi.org/10.1109/WSC.2010.5679125.
4. Ridgway D, Broderick G, Ellison MJ. Accommodating space, time and randomness in network simulation. Curr Opin Biotechnol. 2006;17(5): 493–8. https://doi.org/10.1016/j.copbio.2006.08.004.
5. Takahashi K, Tănase-Nicola S, Wolde PRt. Spatio-temporal correlations can drastically change the response of a MAPK pathway. Proc Natl Acad Sci. 2010;107(6):2473–8. https://doi.org/10.1073/pnas.0906885107.
6. Cowan AE, Moraru II, Schaff JC, Slepchenko BM, Loew LM. Chapter 8 - Spatial Modeling of Cell Signaling Networks. In: Asthagiri AR, Arkin AP, editors. Methods in Cell Biology, Computational Methods in Cell Biology, vol. 110, pp. 192-221. Cambridge: Academic Press; 2012. https://doi.org/10.1016/B978-0-12-388403-9.00008-4. http://www.sciencedirect.com/science/article/pii/B9780123884039000084. Accessed 23 July 2019.
7. Takahashi K, Arjunan SNV, Tomita M. Space in systems biology of signaling pathways - towards intracellular molecular crowding in silico. FEBS Lett. 2005;579(8):1783–8. https://doi.org/10.1016/j.febslet.2005.01.072.
8. Schöneberg J, Ullrich A, Noé F. Simulation tools for particle-based reaction-diffusion dynamics in continuous space. BMC Biophysics. 2014;7(1):11. https://doi.org/10.1186/s13628-014-0011-5.
9. Gillespie DT. Stochastic Simulation of Chemical Kinetics. Annu Rev Phys Chem. 2007;58(1):35–55. https://doi.org/10.1146/annurev.physchem.58.032806.104637. Accessed 23 July 2019.
10. Regev A, Panina EM, Silverman W, Cardelli L, Shapiro E. BioAmbients: an abstraction for biological compartments. Theor Comput Sci. 2004;325(1): 141–67. https://doi.org/10.1016/j.tcs.2004.03.061. Accessed 23 July 2019.
11. Kerr RA, Bartol TM, Kaminsky B, Dittrich M, Chang J-CJ, Baden SB, Sejnowski TJ, Stiles JR. Fast Monte Carlo Simulation Methods For BIOLOGICAL Reaction-Diffusion Systems In Solution And On Surfaces. SIAM J Sci Comput Publ Soc Ind Appl Math. 2008;30(6):3126. https://doi.org/10.1137/070692017. Accessed 23 July 2019.

12. Andrews SS, Addy NJ, Brent R, Arkin AP. Detailed Simulations of Cell Biology with Smoldyn 2.1. PLoS Comput Biol. 2010;6(3):. https://doi.org/10.1371/journal.pcbi.1000705.
13. Blinov ML, Schaff JC, Vasilescu D, Moraru II, Bloom JE, Loew LM. Compartmental and Spatial Rule-Based Modeling with Virtual Cell. Biophys J. 2017;113(7):1365–72. https://doi.org/10.1016/j.bpj.2017.08.022.
14. Hattne J, Fange D, Elf J. Stochastic reaction-diffusion simulation with MesoRD. Bioinformatics. 2005;21(12):2923–4. https://doi.org/10.1093/bioinformatics/bti431. Accessed 23 July 2019.
15. Andrews SS. Smoldyn: particle-based simulation with rule-based modeling, improved molecular interaction and a library interface. Bioinformatics. 2017;33(5):710–7. https://doi.org/10.1093/bioinformatics/btw700.
16. Michalski P, Loew L. SpringSaLaD: A Spatial, Particle-Based Biochemical Simulation Platform with Excluded Volume. Biophys J. 2016;110(3):523–29. https://doi.org/10.1016/j.bpj.2015.12.026.
17. Schöneberg J, Noé F. ReaDDy - A Software for Particle-Based Reaction-Diffusion Dynamics in Crowded Cellular Environments. PLoS ONE. 2013;8(9):74261. https://doi.org/10.1371/journal.pone.0074261.
18. Gruenert G, Ibrahim B, Lenser T, Lohel M, Hinze T, Dittrich P. Rule-based spatial modeling with diffusing, geometrically constrained molecules. BMC Bioinformatics. 2010;11(1):307. https://doi.org/10.1186/1471-2105-11-307.
19. Donovan RM, Tapia J-J, Sullivan DP, Faeder JR, Murphy RF, Dittrich M, Zuckerman DM. Unbiased Rare Event Sampling in Spatial Stochastic Systems Biology Models Using a Weighted Ensemble of Trajectories. PLOS Comput Biol. 2016;12(2):1004611. https://doi.org/10.1371/journal.pcbi.1004611.
20. Bittig AT, Uhrmacher AM. ML-Space: Hybrid Spatial Gillespie and Particle Simulation of Multi-level Rule-based Models in Cell Biology. IEEE/ACM Transactions on Computational Biology and Bioinformatics. 2016;PP(99):1–16. https://doi.org/10.1109/TCBB.2016.2598162.
21. Flegg MB, Chapman SJ, Erban R. The two-regime method for optimizing stochastic reaction-diffusion simulations. J R Soc Interf. 2012;9(70):859–68. https://doi.org/10.1098/rsif.2011.0574.
22. Klann M, Ganguly A, Koeppl H. Hybrid spatial Gillespie and particle tracking simulation. Bioinformatics. 2012;28(18):549–55. https://doi.org/10.1093/bioinformatics/bts384.
23. Kim Y, Stolarska MA, Othmer HG. A hybrid model for tumor spheroid growth in vitro i: theoretical development and early results. Mathematical Models and Methods in Applied Sciences. 2007;17(supp01):1773–98. https://doi.org/10.1142/S0218202507002479. Accessed 23 July 2019.
24. Schaff JC, Gao F, Li Y, Novak IL, Slepchenko BM. Numerical Approach to Spatial Deterministic-Stochastic Models Arising in Cell Biology. PLOS Comput Biol. 2016;12(12):1005236. https://doi.org/10.1371/journal.pcbi.1005236. Accessed 3 May 2018.
25. Hall D, Minton AP. Macromolecular crowding: qualitative and semiquantitative successes, quantitative challenges. Biochim Biophys Acta (BBA) - Protein Proteomics. 2003;1649(2):127–39. https://doi.org/10.1016/S1570-9639(03)00167-5.
26. Neefjes J, Jongsma MML, Berlin I. Stop or Go? Endosome Positioning in the Establishment of Compartment Architecture, Dynamics, and Function. Trends Cell Biol. 2017;27(8):580–94. https://doi.org/10.1016/j.tcb.2017.03.002.
27. Peng D, Warnke T, Haack F, Uhrmacher AM. Reusing simulation experiment specifications to support developing models by successive extension. Simul Model Pract Theory. 2016;68:33–53. https://doi.org/10.1016/j.simpat.2016.07.006.
28. Köster T, Perumalla K, Uhrmacher A. Efficient Simulation of Nested Hollow Sphere Intersections: For Dynamically Nested Compartmental Models in Cell Biology. In: Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation SIGSIM-PADS '17, pp. 173–183. New York: ACM; 2017. https://doi.org/10.1145/3064911.3064920.
29. Faeder JR, Blinov ML, Hlavacek WS. Rule-Based Modeling of Biochemical Systems with BioNetGen. In: Maly, IV, (ed.) Systems Biology. Methods in Molecular Biology pp. 113-167. Totowa: Humana Press; 2009. https://doi.org/10.1007/978-1-59745-525-1_5. Accessed 23 July 2019.
30. Maus C, Rybacki S, Uhrmacher AM. Rule-based multi-level modeling of cell biological systems. BMC Syst Biol. 2011;5(1):166. https://doi.org/10.1186/1752-0509-5-166.
31. John M, Lhoussaine C, Niehren J, Uhrmacher AM. The Attributed Pi-Calculus with Priorities. In: Transactions on Computational Systems Biology XII. Lecture Notes in Computer Science, pp. 13?76. Berlin: Springer; 2010. https://link.springer.com/chapter/10.1007/978-3-642-11712-1_2.
32. Pârvu O., Gilbert D., Heiner M., Liu F., Saunders N., Shaw S. Spatial-Temporal Modelling and Analysis of Bacterial Colonies with Phase Variable Genes. ACM Trans Model Comput Simul. 2015;25(2):13–11325. https://doi.org/10.1145/2742546.
33. Wang Z, Butner JD, Kerketta R, Cristini V, Deisboeck TS. Simulating Cancer Growth with Multiscale Agent-Based Modeling. Semin Cancer Biol. 2015;30:70–8. https://doi.org/10.1016/j.semcancer.2014.04.001.
34. Coffey WT, Kalmykov YP, Waldron JT. The Langevin Equation: With Applications to Stochastic Problems in Physics, Chemistry and Electrical Engineering, Revised. Singapore; River Edge, NJ: World Scientific Pub Co Inc; 2004. https://www.worldscientific.com/worldscibooks/10.1142/8195.
35. Hertz H. Über die Berührung fester elastischer Körper. Journal für die reine und angewandte Mathematik (Crelle's Journal). 1882;92:156–71. https://doi.org/10.1515/crll.1882.92.156.
36. Jeschke M, Ewald R. Large-Scale Design Space Exploration of SSA. In: Computational Methods in Systems Biology, Lecture Notes in Computer Science, pp. 211-230. Rostock: Springer; 2008. https://link.springer.com/chapter/10.1007/978-3-540-88562-7_17.
37. Johnson ME, Hummer G. Free-Propagator Reweighting Integrator for Single-Particle Dynamics in Reaction-Diffusion Models of Heterogeneous Protein-Protein Interaction Systems. Phys Rev X. 2014;4(3):031037. https://doi.org/10.1103/PhysRevX.4.031037.
38. von Smoluchowski M. Versuch einer mathematischen Theorie der Koagulationskinetik kolloider Lösungen. Z für Phys Chem. 1917;92:129–68.
39. Collins FC, Kimball GE. Diffusion-controlled reaction rates. J Colloid Sci. 1949;4(4):425–37. https://doi.org/10.1016/0095-8522(49)90023-9.
40. Noyes RM. Effects of diffusion rates on chemical kinetics. Prog React Kinet. 1961;1:129–60. https://ci.nii.ac.jp/naid/10016743949/. Accessed 12 June 2018.
41. Van Deursen A, Klint P, Visser J. Domain-specific languages: An annotated bibliography. SIGPLAN Not. 2000;35(6):26–36. https://doi.org/10.1145/352029.352035.
42. Zunino R, Nikolic D, Priami C, Kahramanogulları O, Schiavinotto T. I: An Imperative DSL to Stochastically Simulate Biological Systems. In: Bodei, C, Ferrari, G, Priami, C, (eds.) Programming Languages with Applications to Biology and Security: Essays Dedicated to Pierpaolo Degano on the Occasion Of His 65th Birthday, pp. 354-357. Cham: Springer; 2015. https://doi.org/10.1007/978-3-319-25527-9_23.
43. Himmelspach J, Uhrmacher AM. Plug'n Simulate. In: Simulation Symposium, 2007. ANSS '07. 40th Annual; 2007. p. 137–43. https://doi.org/10.1109/ANSS.2007.34.
44. Mannella R. Numerical Stochastic Integration for Quasi-Symplectic Flows. SIAM J Sci Comput. 2006. https://doi.org/10.1137/040620965.
45. The CImg Library - C++ Template Image Processing Toolkit. http://cimg.eu/. Accessed 11 Feb 2019.
46. OpenGL - The Industry Standard for High Performance Graphics. https://www.opengl.org/. Accessed 11 Feb 2019.
47. Yogurtcu O. N., Johnson M. E. Theory of bi-molecular association dynamics in 2d for accurate model and experimental parameterization of binding rates. J Chem Phys. 2015;143(8):084117. https://doi.org/10.1063/1.4929390. Accessed 23 July 2019.
48. Heinrich R, Rapoport TA. Generation of nonidentical compartments in vesicular transport systems. J Cell Biol. 2005;168(2):271–80. https://doi.org/10.1083/jcb.200409087.
49. Tyson JJ. Modeling the cell division cycle: cdc2 and cyclin interactions. Proc Natl Acad Sci. 1991;88(16):7328–32. https://doi.org/10.1073/pnas.88.16.7328.
50. Nicolau DV, Burrage K, Parton RG, Hancock JF. Identifying Optimal Lipid Raft Characteristics Required To Promote Nanoscale Protein-Protein Interactions on the Plasma Membrane. Mol Cell Biol. 2006;26(1):313–23. https://doi.org/10.1128/MCB.26.1.313-323.2006.
51. Haack F, Burrage K, Redmer R, Uhrmacher AM. Studying the Role of Lipid Rafts on Protein Receptor Bindings with Cellular Automata. IEEE/ACM Trans Comput Biol Bioinformatics. 2013;10(3):760–70. https://doi.org/10.1109/TCBB.2013.40.
52. Haack F, Lemcke H, Ewald R, Rharass T, Uhrmacher AM. Spatio-temporal Model of Endogenous ROS and Raft-Dependent WNT/Beta-Catenin Signaling Driving Cell Fate Commitment in Human Neural Progenitor Cells. PLOS Comput Biol. 2015;11(3):1004106. https://doi.org/10.1371/journal.pcbi.1004106.

[C] Performance and Soundness: Case Study of Celluar Automaton for HIV

Cellular automata are often used for spatial dynamics in cell biology such as the spread of infections within a cell biological population. Due to the number of cells in a model, efficient simulation algorithms have received increasing attention over the last decade. Many cellular automata models are executed synchronously. In this paper, we focus on improving the efficiency of a cellular automaton model known as the 'dos Santos' model for the Human Immunodeficiency Virus (HIV). The synchronous updates in this model are driven by mostly deterministic transitions and a few highly probabilistic transitions. This design can be exploited to create an efficient simulation algorithm. We propose such an algorithm using an advanced scheme to generate random numbers (to efficiently perform probabilistic transitions) that efficiently leverages CPU vectorization. The benchmarks show a significant performance increase by a factor of 2.4x in comparison to a quality baseline implementation. We note several limitations in the model behavior and simulation outputs, which apply not only to the dos Santos design but also to the many synchronous HIV models inheriting its design. Several mitigation schemes are discussed to address some of these limitations.

[271] T. Köster, P. J. Giabbanelli, and A. M. Uhrmacher, *Performance and Soundness of Simulation: A Case Study Based on a Cellular Automaton for in-Body Spread of HIV*, in *Winter Simulation Conference (WSC 2020)* (IEEE Computer Society, 2020), pp. 2281–2292

Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

PERFORMANCE AND SOUNDNESS OF SIMULATION: A CASE STUDY BASED ON A CELLULAR AUTOMATON FOR IN-BODY SPREAD OF HIV

Till Köster
Institute for Visual and Analytic Computing
University of Rostock
Germany

Philippe J. Giabbanelli
Dept. of Computer Science & Software Engineering
Miami University, Oxford, OH 45056
USA

Adelinde Uhrmacher
Institute for Visual and Analytic Computing
University of Rostock
Germany

ABSTRACT

Cellular automata are often used for spatial dynamics in cell biology such as the spread of infections within a cell biological population. Due to the number of cells in a model, efficient simulation algorithms have received increasing attention over the last decade. Many cellular automata models are executed synchronously. In this paper, we focus on improving the efficiency of a cellular automaton model known as the 'dos Santos' model for the Human Immunodeficiency Virus (HIV). The synchronous updates in this model are driven by mostly deterministic transitions and a few highly probabilistic transitions. This design can be exploited to create an efficient simulation algorithm. We propose such an algorithm using an advanced scheme to generate random numbers (to efficiently perform probabilistic transitions) that efficiently leverages CPU vectorization. The benchmarks show a significant performance increase by a factor of 2.4x in comparison to a quality baseline implementation. We note several limitations in the model behavior and simulation outputs, which apply not only to the dos Santos design but also to the many synchronous HIV models inheriting its design. Several mitigation schemes are discussed to address some of these limitations.

1 Introduction

Over 30 million people have died from complications associated with HIV and AIDS. By the end of 2018, it is estimated that between 32.7 and 44.0 million were living with HIV (World Health Organization 2020). Modeling & Simulation (M&S) has been used in HIV research for several decades to understand the spread of HIV (both within the body and across individuals) or evaluate the potential outcomes associated with interventions. M&S is particularly useful as a first step in intervention design as it allows to predict effects in a 'virtual laboratory' without causing harm to real-world individuals. Two successful modeling projects include Kaplan's early work, which contributed to distributing clean needles to injection drug users (Kaplan 1989), and the study directed by Granich, which strengthened the evidence base for universal testing and immediate therapy (Granich et al. 2009).

Cellular Automata (CA) provide one of the approaches used to simulate HIV. This approach is most commonly used to model the spread within the body, as the regular packing of cells within CA provides an analogy to the tight packing of immune cells in lymphoid tissues which are targeted by the virus. Although several CA models of the immune response to the virus were proposed in the early 1990s (Pandey and Stauffer 1990; Kougias and Schulte 1990), this approach gained attention with the model in (dos Santos and Coutinho 2001), now commonly called the 'dos Santos model'. Its popularity is partly due to its ability to replicate the phases of HIV infection (i.e. before the onset of AIDS) despite its apparent simplicity.

Since many stochastic in-body HIV CA models extend the structure of the dos Santos model (Precharattana 2016), investigating its performance and soundness allows to examine the broader field of HIV CA. As the simple implementations prevalent in the field are very time intensive to run, simulations of these stochastic models often use too few runs to yield tight confidence intervals, hence results may not be entirely replicated in future studies (Giabbanelli et al. 2019). Although these models succeed at producing realistic average curves for viral load, we may be *obtaining the right results for the wrong reasons* since the models are driven by very high rates of cell-to-cell infection (i.e. between close neighbors) unlike the real-world mix of cell-to-cell *and* cell-free infections via the bloodstream (Giabbanelli et al. 2019). Recent studies demonstrate performances of the dos Santos model can be improved through just in time compilation of Python code (Giabbanelli et al. 2020) or by using machine learning meta-models (Fisher et al. 2020). To further improve upon these results, we need to study the model's performance characteristics.

Our study builds on previous work in two ways:

1. We increase performance by a factor of 2.4 by performing the first in-depth performance study for the dos Santos CA model and using algorithmic optimizations in a modern native language (Rust).
2. We examine whether the design of CA models (particularly the synchronous transitions and their high probabilities) produces behavior that differs from real-world expectations.

The paper is structured into four parts. In Section 2, we provide a brief background on the design and implementation of the biological CA models, including the dos Santos model. In Section 3, we introduce our approach to profile an optimization. Based on our analysis of the technical properties of the implementation, we propose algorithmic performance improvements in section 4. Finally, we reflect on the soundness of the overall approach.

2 Background: Design and Optimization of CA Models in Cell Biology

A cellular automaton is a discrete model in three ways: cells are updated at discrete time steps, the space is discretized into a regular tiling (such as a grid or a hexagonal tiling), and cells are in discrete states. The rules to update the cells can involve probabilities (hence creating a *stochastic* model), time (e.g., infected cells will die at the next step), or the states of neighbors (e.g. being next to an infected cell makes it possible to pass on virions). Models without probabilistic transitions are *deterministic*, while those with at least one such transition are *stochastic*. All CA for HIV are *synchronous*, which means that cells are all updated at the same time by using a copy to store their future states before committing them.

CA used to model other biological systems, such as cancer growth, typically use *asynchronous* synchronization (Deutsch et al. 2005; Metzcar et al. 2019). This introduces new possibilities for optimization, such as finding efficient means to iterate through the cells or selecting a random neighbor (Poleszczuk and Enderling 2014). A common approach to improving performance for cellular automata is to leverage their 'embarrassingly parallel' update step. Parallelization has been done many times in CA (Salguero et al. 2019), including by running them over GPU-based implementations (Rybacki et al. 2009). Other improvements have leveraged the trade-off between runtime performance and memory footprint, for instance by using memoization algorithms that exploit spatial regularities (Gosper 1984; Caux et al. 2010).

The dos Santos model and its successors stand out among biological models for three defining characteristics: transitions are mostly step-based, very high probabilities are used in stochastic transitions, and only the transition from 'healthy' (or variations thereof) to 'infected' involves neighbors. The structure of the dos Santos model (Figure 1) consists of four main states (healthy, infected A_1, infected A_2, dead) where deterministic transitions occur either at constant time intervals (e.g., A_1 turns into A_2 after 4 steps, A_2 turns into dead in the next step) or are driven by very high probabilities (e.g., dead cells are replenished by the system with 99% probability). The dos Santos model has inspired many derived models, which often consist of adding states and accompanying transitions to explore key concepts such as viral reservoirs (Shi et al. 2008), therapy (González et al. 2013), or adherence to treatment (Rana et al. 2015).

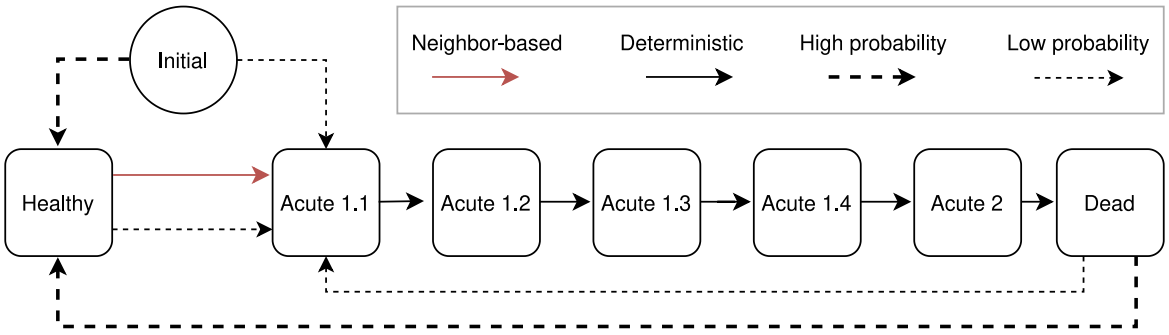


Figure 1: The dos Santos model has 4 states (healthy, infected A_1 and A_2, dead). As A_1 turns A_2 over four time steps, we break it down into four consecutive states, resulting in 7 states. Note that most transitions are either deterministic (plain line) or have a very high probability (broad dashed line).

This 'line of models' perpetuates several design *choices* such as step-based transitions and tends to be faithful to the initial model even in the parameter values. For instance, the dos Santos model uses 4 steps to go from A_1 to A_2. This was still the case for models published up to 15 years later (González et al. 2013; Rana et al. 2015; Precharattana et al. 2010; Moonchai and Lenbury 2016). Similarly, the 99% probability of replacing a dead cell by a healthy one was still in effect 16 years later (Golpayegani et al. 2017).

Although some HIV CA models are occasionally created entirely from scratch and hence bear no resemble to the dos Santos model (Hillmann et al. 2017), the fact that a robust line of research inherits the dos Santos design and parameter values suggests that our findings can generalize to a *class* of HIV models.

3 Understanding the Efficiency via Profiling

We have created an implementation with instrumentation of the model as well as created multiple synthetic benchmarks to investigate specific aspects (e.g. memory limitations) of the algorithm. The implementation is done in Rust, a modern native language. For the performance measurements we use the Criterion library (Version 0.3). Measurements were done on a Workstation with an Intel Xeon E5-1630 v4 CPU using Rust 1.41. The code, benchmarks, and scripts to produce the plots can be found at https://osf.io/g5yah/.

3.1 Storing and Iterating Over Model States

Each cell in the model may be in one of 7 states (Healthy, Acute with $\tau \in \{0,1,2,3\}$, Latent, Dead). We store these states in one byte. Although fewer bits may be used, the small gain in space would be at the expense of increased computations to pack/unpack states. To support compiler-based optimizations, we store states in a contiguous (one dimension) array, which supports fast read/write operations.

Vectorization is an optimization where a program uses CPU specific instructions that can handle multiple data chunks in one step. This is a type of instruction level parallization also known as Single Instruction Multiple Data (SIMD). For improvements in the vectorization we use the ndarray (version 0.13) library which provides a matrix abstraction that we use for the implementation of our algorithms. Readers more familiar with Python than with Rust may think of this abstraction as similar to numpy, albeit more closely integrated into the language. We use the ndarray zip operator to iterate over the grid. Neighborhood checks are done via iterations based on indexes[1].

Note that this baseline implementation is not a naive one, as that would lead to unrealistically low performances. Our baseline implements a CA with several techniques including making an efficient use of the memory and optimizing the iteration schemes.

[1]By default this would not necessarily be fast in Rust. All typical array access via indices are bound-checked at runtime. This was circumvented for this implementation by using unsafe code.

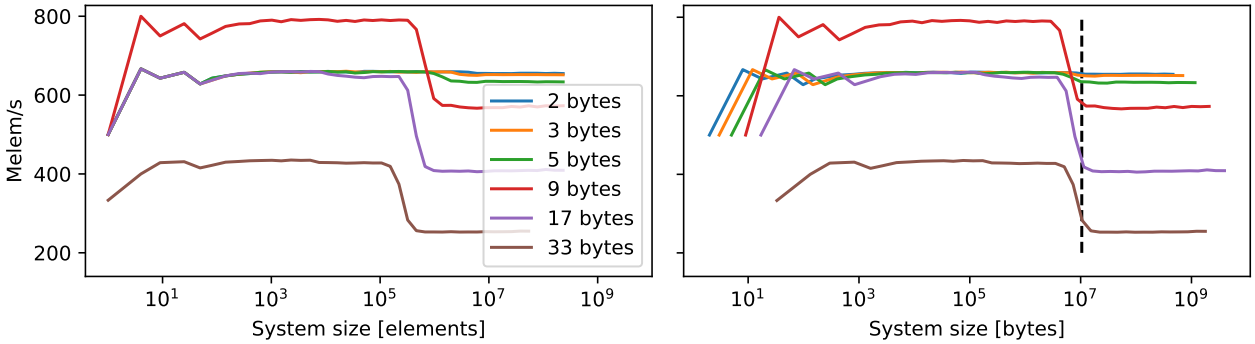


Figure 2: Throughput of different synthetic benchmarks for different number of cells. The different lines correspond to different data sizes. There is a different peak throughput due to different computational optimization, with 64bit integers shown in red as fastest due to the optimal native support and 256 bit integers requiring two separate operations. For some larger per cell memory needs we can see how CPU caches are no longer able to provide a performance boost when considering larger number of cells. For smaller per cell storage however even RAM memory rates are sufficient. The separation at 10MB (vertical dashed line) is consistent with the size of this CPUs L3 Cache. These results also indicate that our results are applicable to arbitrarily large systems as well.

3.2 Performance Metric

Our performance metric is the number of cells processed per second, given in million elements per second (Melem/s). We will be measuring this throughput at each step, as well as averaged across the whole 600 steps that are commonly conducted in an HIV CA simulation. This metric only accounts for the operation that are required in the simulation model, hence additional operations used for analyses (e.g., tracking the number of cells in each state) are not included in our performance metric.

3.3 Synthetic Benchmark and Maximum Theoretical Throughput

In this section we successively build up some synthetic benchmarks, to get an estimate for potential performance limitations. Firstly, to identify the maximum throughput that our machine can achieve, we create a synthetic benchmark consisting of an array randomly filled with states $S_1, S_2, ...$, which we replace using simple rules such as $S_1 \to S_5$ and $S_5 \to S_2$. This is the most minimal and simple version of this type of simulation model and implementation. The throughput we get for this benchmark is the limit no traditional implementation of an HIV model could surpass, since such models always involve more computations in their transitions (e.g., accessing neighbors, applying probabilities) Note that there are more advanced works that do not need to access the entire system, like Memoization (Gosper 1984), but those are beyond the scope of this paper.

As this benchmark involves very few 'hard' operations in contrast to HIV models, its throughput may be too high. We thus create a second benchmark to offer a tighter bound, by incorporating harder operations. Specifically, we make optimal code generation a little harder for the compiler by introducing a state S_{null}. This state is not created initially and none of the other states transitions to it, but its transition rule is very complex requiring change of a global state, similar to a random number generator or a counting mechanism. This limits the kind of optimizations a compiler and CPU can perform, like vectorization.

This already has a tremendous influence on the performance. For the first benchmark we get a rate of 788 Melem/s whereas the second yields only 156 Melem/s. We should stress again that the only difference between these two benchmarks is a state transition rule, which is never triggered but whose conditions make it harder to optimize the code..

For the purpose of comparison we have created a third synthetic benchmark with all identical cells, applying the same hard to vectorize kernel that yielded 156 Melem/s earlier we now get more than 1470 Melem/s. This is because the CPU can optimize at runtime for identical data. The increased mixing of the system therefore leads to slower performance.

A few of the transitions in the dos Santos model depend on probabilities. To understand the impact of these stochastic transitions on performances, we consider an extreme case in which half of all transitions are stochastic in a fourth and fifth benchmark. We modify the first synthetic benchmark model accordingly. Naturally this raises the question of how the random numbers are generated. We consider two cases, one with Rusts default, very high quality cryptographically secure ChaCha block cipher random number generator and one using the much faster Xoroshift random number generator. For these model implementations we get rates of 83 Melem/s and 134 Melem/s respectively.

The difference between the easy and hard to vectorize throughput (first and second) is already an indication that memory access is likely not the limiting factor for the performance of this computational problem (i.e. it is compute- rather than memory-bound). To confirm this we run a series of tests where we modify our first synthetic benchmark as such: In between the states of the dummy model we interweave counters that are incremented on every access. We use different data types to test for the memory limit. Results (Figure 2) show the effect of caches for the smaller grid sizes, but for a larger amount of cells and larger counters we run into the memory limit. This limit however is very far from the domain applicable here, as our memory load is very small. For any real model implementation, the transition rules are more complex and therefore putting even less strain on memory throughout. Therefore in the following we can focus on optimization for computational throughput.

3.4 Results for our baseline implementation

Our baseline implementation has a throughput of about 76 Melem/s, which is about 10% of the maximum as established by the first synthetic benchmark (Figure 2), and already about half of the more reasonable harder to vectorize second benchmark.

The throughput of the baseline implementation changes throughout the model run. After the first few steps (which show extreme model behavior), performance increases, then after about 200 steps decreases. We can explain this by looking at the simulation behavior (Figure 3). Initially, we have many healthy cells (Figure 3(a)–top), each of which requires an expensive neighborhood check. Later, there are fewer healthy cells, hence performances improve. However, after about 220 steps, performances degrade again even though the number of healthy cells (as well as all other cells) remains constant. This behavior can be understood by examining the heterogeneity in the simulation (Figure 3(b)). Initially the system is quite well ordered and homogeneous, as measured by the percentage of cells whose left neighbor has the same state. These are particularly easy to execute efficiently by the CPU.

Using faster (but lower quality) Xorshift random numbers leads to a throughput of about 91 Melem/s in comparison to 76 Melem/s before.

4 Algorithmic Performance Improvements

Results in the previous section have established that randomness and branching intensive code (e.g. neighborhood checks) are the limiting factors for performances in our implementation. In this section, we present three optimizations to improve performances: avoiding the over-generation of random numbers, reducing the number of conditional checks when looking at the states of neighboring cells, and a combination.

4.1 Random Number Calculation via Geometrical Distribution

In the traditional implementation of this CA model, the random numbers are sampled for every cell. For example, *every* dead cell requires the computation of a probability p_H to decide whether it is replaced with a healthy one. Our analysis of the rates used specifically in this model (Figure 1) reveal that stochastic events

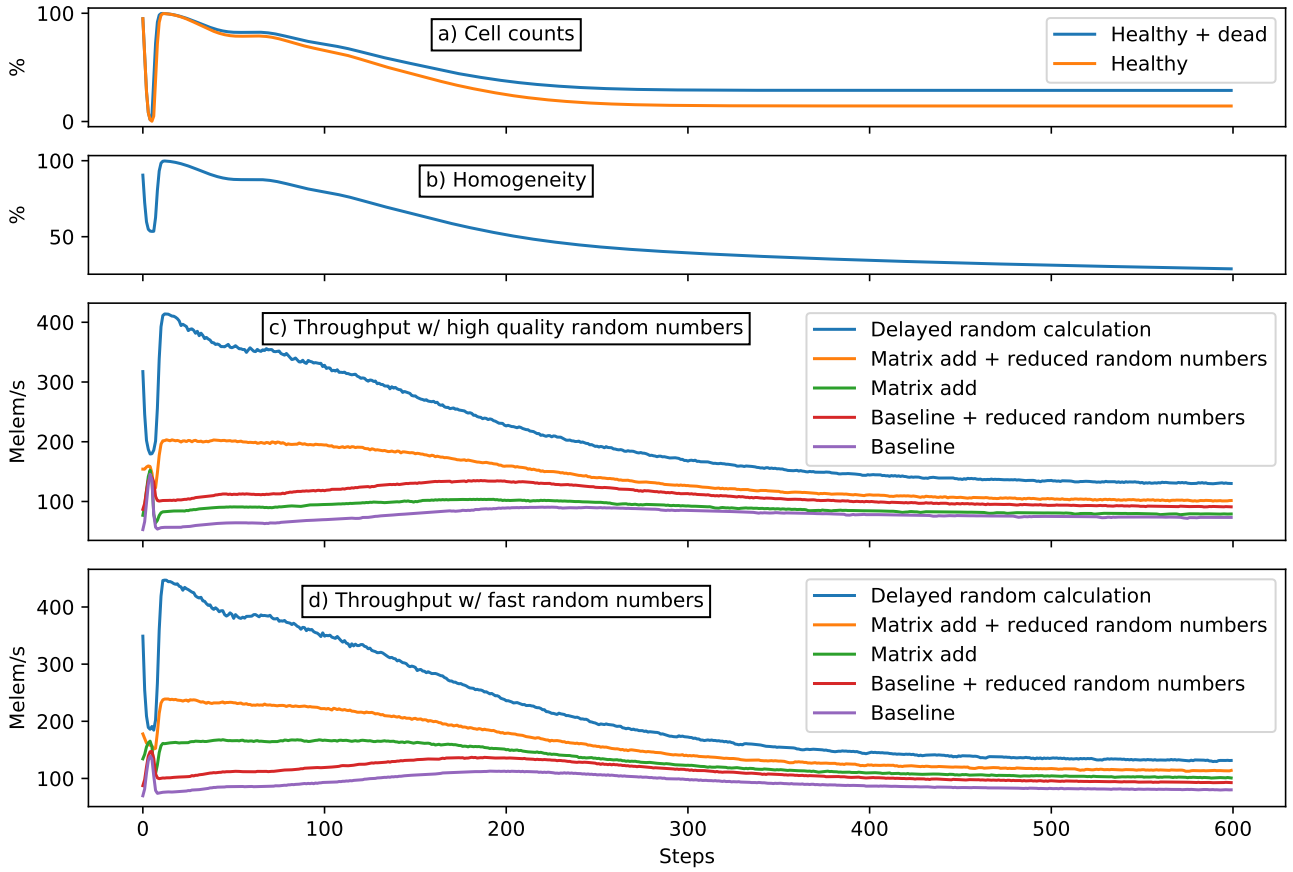


Figure 3: The simulator output is shown by counting healthy and dead cells (a) and the number of cells that have the same state as the cell to their left (b). The throughput is shown in two configurations: very high quality random numbers (c), or low quality but faster to compute (d). All runs use 10 replications and a grid of 1500×1500 cells. Step-averaged data including errors can be found in Table 1.

are either very rare (probability is close to zero) or very likely (probability almost one). Calculating the random probability for every cell is very costly. To lower this cost, we note that events are all independent, but follow the same Bernoulli process. For example if we have an event with probability $= 0.997$, a negative sample will occur on average only about every 333 samples. The number of negative samples between two positive samples follows the Geometric distribution. We only need to calculate one sample of that distribution (using a standard library) and then count until we have reached this number of occurrences. This works well for both very high as well as very low probabilities, as for those the number of avoided random number calculations is the highest.

4.2 Neighborhood checks via Matrix addition

Encoding the neighborhood checks using addition instead of iteration and equality checks has already been presented in the literature (Giabbanelli et al. 2020). However, these checks were still performed on demand. That is, a cell was first checked to know whether it was healthy, and then the addition was performed.

In the early phase of the simulation, most cells are healthy. To calculate the newly infected cells, all cells in the neighborhood of each cell are being tested. To optimize this calculation, we introduce vectorized additions. For that we take the entire grid as a matrix and add the eight shifted variants (for the Moore neighborhood) and add them to this matrix. These additions are very cheap, as we can use highly optimized (and vectorized) linear algebra libraries.

Random Numbers	Simulator	prob. transitions	throughput [Melem/s]	setup in %
high quality	baseline	every time	76.4 ± 0.1	
		geometric distr.	108.0 ± 0.2	
	matrix add	every time	88.9 ± 0.5	9.8 ± 0.5
		geometric distr.	131.8 ± 1.1	14.5 ± 0.7
	delayed random	n/a	182 ± 3	27.6 ± 0.7
high speed	baseline	every time	91.1 ± 0.1	
		geometric distr.	109.5 ± 0.2	
	matrix add	every time	126 ± 1	13.7 ± 0.7
		geometric distr.	148 ± 2	16.5 ± 0.7
	delayed random	n/a	186 ± 2	24.6 ± 0.8

Table 1: Average throughput of the different simulators and their configurations configurations. Setup time is the time doing matrix adding as well as the random transition for the delayed random variant. Taken from 10 replications at 1500x1500 cells with errors showing standard error of mean. The behavior of efficiency through the model run can be found in Figure 3.

Later on we can iterate over the entire grid without the need to use any complex access patterns, as we have already computed the neighborhood for every cell, which minimizes branching. As observed in the synthetic benchmarks, branching is detrimental to optimal throughput. In branched code, the CPU has struggles with branch prediction and vectorization can be less efficient. This optimization is expected to be the most helpful in the first third of the simulation where there still are many healthy cells.

4.3 Delayed random transitions

Our third optimization in a way combines the first two presented. To improve vectorization and execution speed we need to minimize branching. One type of branching we have is checking at the random transitions. Even if we do not need to compute a new random number (using the geometric distribution) we still might need to alter a counter and check if it is equal to zero.

In this third optimization we remove all stochastic transitions by rounding the probabilities to one or zero. Together with the matrix addition optimization outlined above, this implementation has almost no branching. The only check that remains is in the presence of a healthy cell, as we need to check whether the pre-computed neighborhood sum exceeds the threshold.

Simply removing stochastic transitions would significantly alter model behavior. Therefore we need to find a way to reintroduce these rare events where stochastic tests would have failed. Here we leverage that the number of times such a test passes or fails follows the binomial distribution. After a step has been executed, we then visit a number of cells sampled from the correct binomial distribution and reverse the step. This binomial distribution takes the probability of success per cell and the total number of cells as parameters. Therefore the optimization is still exact and not approximate.

4.4 Results for the Proposed Three Optimizations

We have implemented the optimizations presented above and the results can be found per step in Figure 3 and in step averaged form in Table 1.

Reducing the number of random numbers improves the baseline speed by 40% if high quality random number generators are used and by 20% if the fast random number generators are used. As expected this optimization mainly affects the beginning of the simulation where healthy and dead cells (i.e. those that need random numbers) make up a majority of the cells. When using the high quality random numbers, this optimization even outperforms the matrix add optimization.

Using the matrix addition for the environment checks also improves performance with throughput rates of up to 148 Melem/s. Again this also mainly affects the first third of the simulation where there are very many healthy cells that need to check their environment. In the later stages of the simulation this optimization only slightly improves upon the baseline. Performing this matrix addition takes only about 15% of the runtime, thus it is not a large contribution to runtime, even though potentially more work than is strictly needed is performed.

The delayed random calculation where random transitions are added afterwards is again a lot faster than the optimizations presented above. It reaches speeds in excess of 400 Melem/s leading to an average throughput of about 186 Melem/s. Performance mainly goes down with increased mixing in the system. Performing the delayed random calculation as well as using matrix addition for the testing the neighborhood takes about 25% of the total runtime, so the main iteration loop still dominates.

Overall, we see a 2.4 speed up, when comparing the fastest average throughput of 182 Melem/s with the baseline of 76.4 Melem/s for high quality random numbers.

4.5 Alternative Approaches

Modeling studies rarely report *negative* results, and HIV CA studies are no exception. However, it is important to know which other approaches could be attempted and the results that they lead to. Some of the results are negative *in our context* (e.g., no throughout improvements) but would be of interest in other cases (e.g., better bandwidth utilization). By disclosing these alternatives, we thus limit the risk that future teams will devote energy to approaches that were not fruitful. Other teams may be interested in some of the approaches below as they do lead to improvements other than throughput, which is the main consideration for this paper.

As the model has mostly a single path of transition (*Healthy* then various levels of *Infected* then *Dead*), we implemented a **buffer free** variant of the simulation. However this has only reduced memory and bandwidth usage and neither of them are currently limiting performances. In practice, this sometimes even reduced performances due to a more complex environment check.

We have examined whether vectorization could be applied locally at the neighborhood level, i.e. **vectorizing the neighborhood check** every time we encounter a healthy cell. However, due to the small neighborhood tested (a Moore of 8 neighbors) as well as the irregularity of the pattern alignment in memory, this decreased performances. Note that there have been attempts to expand the neighborhood of HIV CA models to account for both proximal (cell-to-cell) and more distal (cell-free) infections (Giabbanelli et al. 2019). In such cases, the larger neighborhoods may be better candidates for explicit vectorization.

The dos Santos model seeks to replicate the first two phases of HIV in the absence of any treatment. Individuals thus necessarily have a larger proportion of infected cells later in the simulation (which would not necessarily be the case if therapy was modeled). Given that almost all neighborhoods contain an acutely infected cell in later parts of the dos Santos model run, we assessed the potential of optimizing the neighborhood look-up. For that, we created a **lookup data structure** that was a quarter of the size of the system, with each entry containing the sum of four cells. By checking in the lookup structure first, most neighborhood iterations could be avoided, however accessing the structure itself had a similar overhead, resulting in no speedup. This does not check the neighborhood explicitly, but if any of the four cells contain the an infectious cell, then checking becomes unnecessary.

5 Soundness of the CA Approach

The performance of simulation algorithms is important as an efficient simulation algorithm is necessary to conduct extensive experiments with a simulation model. Designed Experiments help us understand how a model responds or identify which (combinations of) parameter values are most important. By shedding light on the behavior of the model, such experiments contribute to building trust in the model's ability to provide useful answers for its intended application context.

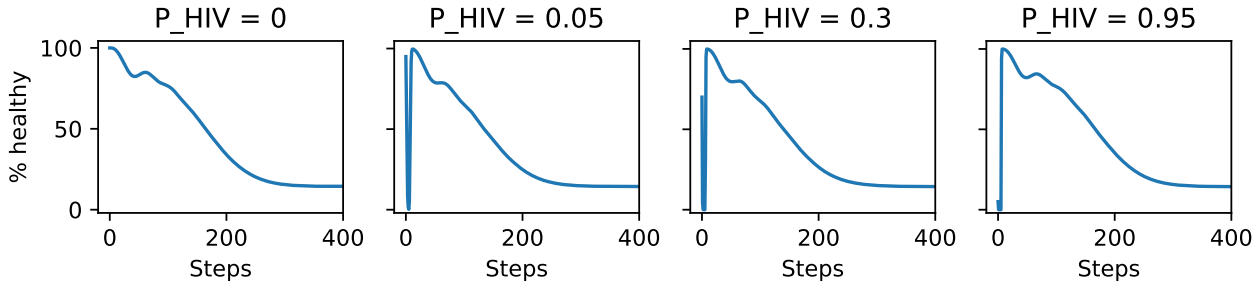


Figure 4: A change in the parameter P_{HIV} only affects the first few steps of the model. This is an artefact of the methodology.

Another essential expectation to build trust is that the simulation algorithm correctly implements the approach. However, another aspect of soundness is whether the approach is appropriate for the system and application context. The former requires comparing the simulation results of different simulation algorithms, which forms an intrinsic part of any performance study, independently of whether exact or approximate, spatial or non-spatial simulation algorithms are evaluated(Jeschke et al. 2011).

It is well known that synchronously updated cellular automata are problematic for many real world simulation applications (Schönfisch and de Roos 1998). Therefore, cellular automata models that simulate tumor growth use an asynchronous updating scheme instead: "These methods [cellular automata] usually update the lattice sites in a random order to reduce grid artifacts" (Metzcar et al. 2019). This is in contrast with the dos Santos model, where (emphasis added) "in one time step the entire lattice is *updated in a synchronized* parallel way" (dos Santos and Coutinho 2001). If a cellular automaton with a regular grid is executed synchronously (as is the case with the dos Santos line), some directions are favored over others. This leads to the known problem of anisotropic behavior (Schönfisch 1995): a direction-dependent speed of information spread throughout the system, usually favoring diagonals. In addition, fixed time steps in simulations as typical for cellular automata execution can introduce further errors. Therefore, fixed time steps should usually only be used if the system itself is one of regular steps. A discussion can be found in the literature, for example in Appendix 1A of (Law 2007).

5.1 Manifestation in this particular model

We now examine the particularities of this specific model (and its close relatives) to investigate whether some of these problems also apply here. The behavior of our model consists of 3 *Phases*. Initially the entire system goes through the complete cycle of infected, dead, and then again healthy, since about one in twenty cells is infected. Here we already have a particular artefact of the synchronous update, that is, that this spread basically dies out immediately. If we were to alter this initial infection proportion from 5% to 95% or 0% the model behavior after the first few step would not change at all. The rest of the trajectories remain the same. This can be seen in Figure 4.

Then there is a second phase, of regular patterns in the system. Here some cells randomly turn sick. Taking the infected cell as origin, waves of infection propagate through the system. However due to the mostly deterministic nature of the transition, the cells all get healthy again after one iteration of the cycle as shown in Figure 1. This is because the waves are very stable and therefore the dead cells form a barrier shielding the healthy cells from becoming infected again. These transitions are rarely avoided due to stochastic sampling (a fact which we leveraged for performance improvements).

On the very rare occasion, a transition does not take place multiple steps in a row, i.e. a cell remains healthy while its surroundings are infected. In this situation, we transition into the third phase, the random noise phase. This cell will be infected at the same time as its surrounding cells become healthy. Those cells in turn get infected, turn this particular cell healthy and so on, therefore the infection cannot die out

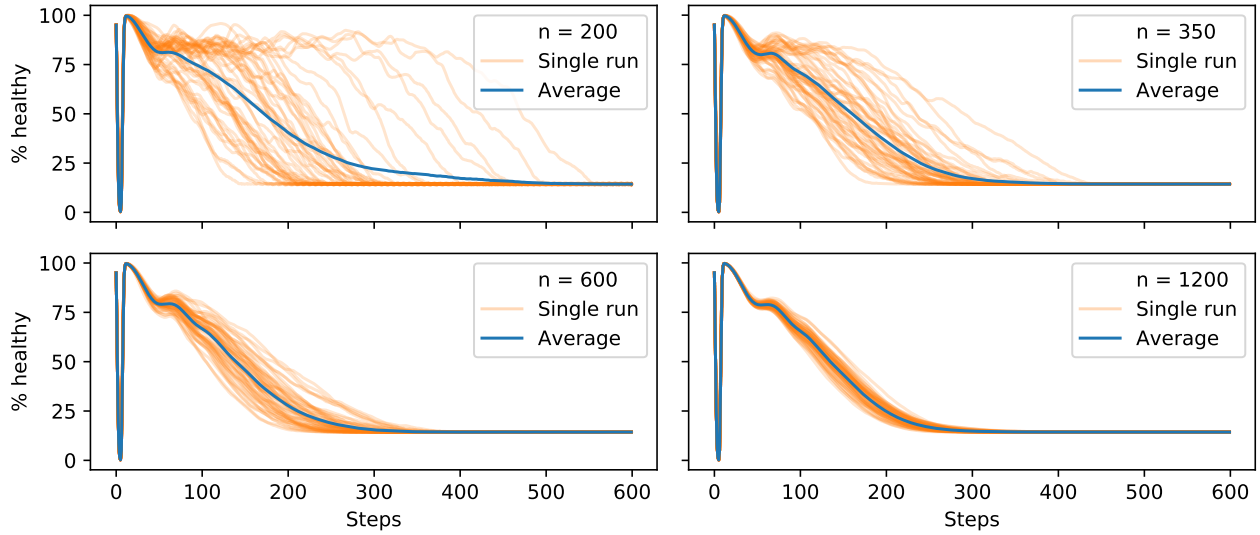


Figure 5: Larger system sizes do not change behavior, but lead to a convergence of individual runs due to spatial averaging. We compare a grid of $n \times n$ cells varying n between 200 and 1200. The percentage of healthy cells is plotted both for the individual runs as well as their average. For larger systems we only see more of a regression to the mean, but no additional insight into the model behavior. At least for the basic dos Santos model, larger models do not provide additional insight.

at this point. This is again an artefact of the synchronous update with high probabilities. This artefact will then serve as a contiguous source of infection waves in the system, forming irregular shapes and leading to a more and more random phase, which is the steady state. This artefact has already been described by the original authors.

Another question is how sensitive the behavior of the model is to changes in its size. The system is simulated for 600 steps. For larger systems, such as 1000x1000, a transition at one end of the system cannot affect a cell more than 600 cells away, since it has a maximum reach of 1 grid cell per time-step. Here, increasing the system size serves to average multiple potential trajectories. A system of 600x600 cells with periodic boundary condition will have the exact same average behavior as a system with 60000x60000 cells.Our performance results however, will be applicable to larger models as well, since our implementation is compute bound. This is the reason we supply throughput and not total runtime numbers.

5.2 Discrete event-based simulation as a solution to problems of soundness

Discrete event-based simulation appears as a natural remedy to counterbalance the problems introduced due to synchronous, step-wise computation of the cellular automata. Approaches such as Cell-DEVS and similar (Zeigler 1982; Wainer 2014) exploit the potential of discrete event simulation for cellular automata type models. They offer the possibility to explicitly associate states with sojourn times, and thus have a natural notion of time and can easily be adapted to arbitrary networks beyond grids. We have tried to implement the dos Santos Model as a discrete event simulation using exponential distributed sojourn times as well as explicit delays. Although the conceptual translation appears straightforward, the resulting model behavior differs significantly. Instead of the initial infection immediately dying out, we transition directly to the stable random state phase, where we have random mixing of all model states. Regular structures that shield infected cells from becoming sick do not occur when allowing transitions in continuous time.

Another indication that the observed behavior is the result of the synchronous, step-wise update is the question what happens if we reduce the step-size. One would expect if the time step is reduced from one week to one day and the probabilities per step accordingly adapted to receive the same results. However,

the smaller probabilities per time step allow for the wave fronts to be interrupted. As a consequence they dissolve quickly. Thus the second phase of the observed behavior will not occur.

6 Conclusion

We assessed the execution characteristics of the established dos Santos cellular automaton model from a performance standpoint. We implemented and tested several algorithmic variants that surpass the performance of an already optimized implementation for these type of CA models by a factor of up to 2.4. A crucial ingredient of the achieved speedup has been separating the model into a small stochastic part and a large deterministic part. The expensive deterministic part maps well onto vector execution of the CPU for this compute bound problem.

Analyzing the behavior of the model revealed several limitations. Retrospectively we observe, that the very characteristics we exploited to increase performance (synchronicity of execution with very rare stochastic variations within the model) are the same ones that also limits the soundness of the approach. Consequently, our novel performance study also emphasizes the importance of analyzing model characteristics and soundness not only of the algorithm but also of the approach.

REFERENCES

Caux, J., P. Siregar, and D. Hill. 2010. "Hash-life algorithm on 3D excitable medium application to integrative biology". In *Proceedings of the 2010 Summer Computer Simulation Conference*, 387–393. Society for Computer Simulation International.

Deutsch, A., S. Dormann et al. 2005. *Cellular automaton modeling of biological pattern formation*. Springer.

dos Santos, R. M. Z., and S. Coutinho. 2001. "Dynamics of HIV infection: A cellular automata approach". *Physical review letters* 87(16):168102.

Fisher, A., B. Adhikari, C. Chai, J. E. Morgan, V. K. Mago, and P. J. Giabbanelli. 2020. "Predicting the resource needs and outcomes of computationally intensive biological simulations". In *Proceedings of the 2020 Spring Simulation Conference*.

Giabbanelli, P. J., C. Freeman, J. A. Devita, T. Koster, and J. A. Kohrt. 2020. "Optimizing Discrete Simulations of the Spread of HIV-1 to Handle Billions of Cells on aWorkstation". In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*.

Giabbanelli, P. J., C. Freeman, J. A. Devita, N. Rosso, and Z. L. Brumme. 2019. "Mechanisms for Cell-to-cell and Cell-free Spread of HIV-1 in Cellular Automata Models". In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 103–114.

Golpayegani, G. N., A. H. Jafari, and N. J. Dabanloo. 2017. "Providing a therapeutic scheduling for HIV infected individuals with genetic algorithms using a cellular automata model of HIV infection in the peripheral blood stream". *Journal of Biomedical Science and Engineering* 10(3):77–106.

González, R. E., S. Coutinho, R. M. Z. dos Santos, and P. H. de Figueirêdo. 2013. "Dynamics of the HIV infection under antiretroviral therapy: A cellular automata approach". *Physica A: Statistical Mechanics and its Applications* 392(19):4701–4716.

Gosper, R. 1984. "Exploiting regularities in large cellular spaces". *Physica D: Nonlinear Phenomena* 10(1):75 – 80.

Granich, R. M., C. F. Gilks, C. Dye, K. M. De Cock, and B. G. Williams. 2009. "Universal voluntary HIV testing with immediate antiretroviral therapy as a strategy for elimination of HIV transmission: a mathematical model". *The Lancet* 373(9657):48–57.

Hillmann, A., M. Crane, and H. J. Ruskin. 2017. "A Computational Lymph Tissue Model for Long Term HIV Infection Progression and Immune Fitness.". In *AICS*, 245–257.

Jeschke, M., R. Ewald, and A. M. Uhrmacher. 2011. "Exploring the performance of spatial stochastic simulation algorithms". *Journal of Computational Physics* 230(7):2562–2574.

Kaplan, E. H. 1989. "Needles that kill: modeling human immunodeficiency virus transmission via shared drug injection equipment in shooting galleries". *Reviews of infectious diseases* 11(2):289–298.

Kougias, C. F., and J. Schulte. 1990. "Simulating the immune response to the HIV-1 virus with cellular automata". *Journal of Statistical Physics* 60(1-2):263–273.

Law, A. M. 2007. *Simulation Modeling & Analysis*. 4 ed. New York, NY, USA: McGraw-Hill.

Metzcar, J., Y. Wang, R. Heiland, and P. Macklin. 2019, November. "A Review of Cell-Based Computational Modeling in Cancer Biology". *JCO Clinical Cancer Informatics* (3):1–13.

Moonchai, S., and Y. Lenbury. 2016. "Investigating Combined Drug and Plasma Apheresis Therapy of HIV Infection by Double Compartment Cellular Automata Simulation". *International Journal of Computer Theory and Engineering* 8(3):190.

Pandey, R. B., and D. Stauffer. 1990. "Metastability with probabilistic cellular automata in an HIV infection". *Journal of statistical physics* 61(1-2):235–240.

Poleszczuk, J., and H. Enderling. 2014. "A High-Performance Cellular Automaton Model of Tumor Growth with Dynamically Growing Domains". *Applied Mathematics* 05(01):144–152.

Precharattana, M. 2016. "Stochastic modeling for dynamics of HIV-1 infection using cellular automata: A review". *Journal of bioinformatics and computational biology* 14(01):1630001.

Precharattana, M., W. Triampo, C. Modchang, D. Triampo, and Y. Lenbury. 2010. "Investigation of spatial pattern formation involving CD4+ T cells in HIV/AIDS dynamics by a Stochastic cellular automata model". *International Journal of Mathematics and Computers in Simulation* 4(4).

Rana, E., P. J. Giabbanelli, N. H. Balabhadrapathruni, X. Li, and V. K. Mago. 2015. "Exploring the relationship between adherence to treatment and viral load through a new discrete simulation model of HIV infectivity". In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 145–156.

Rybacki, S., J. Himmelspach, and A. M. Uhrmacher. 2009. "Experiments with single core, multi-core, and GPU based computation of cellular automata". In *2009 first international conference on advances in system simulation*, 62–67. IEEE.

Salguero, A. G., M. I. Capel, and A. J. Tomeu. 2019. "Parallel Cellular Automaton Tumor Growth Model". In *Practical Applications of Computational Biology and Bioinformatics, 12th International Conference*, edited by F. Fdez-Riverola, M. S. Mohamad, M. Rocha, J. F. De Paz, and P. González, 175–182. Cham: Springer International Publishing.

Schönfisch, B. 1995, February. "Propagation of fronts in cellular automata". *Physica D: Nonlinear Phenomena* 80(4):433–450.

Schönfisch, B., and A. de Roos. 1998. "Synchronous and Asynchronous Updating in Cellular Automata". In *Cellular Automata: Research Towards Industry*, edited by S. Bandini, R. Serra, and F. S. Liverani, 42–46. London: Springer London.

Shi, V., A. Tridane, and Y. Kuang. 2008. "A viral load-based cellular automata approach to modeling HIV dynamics and drug treatment". *Journal of theoretical biology* 253(1):24–35.

Wainer, G. A. 2014. "Cellular Modeling with Cell-DEVS: A Discrete-Event Cellular Automata Formalism". In *Cellular Automata*, 6–15. Cham: Springer International Publishing.

World Health Organization 2020. "Global Health Observatory (GHO) data".

Zeigler, B. P. 1982, June. "Discrete event models for cell space simulation". *International Journal of Theoretical Physics* 21(6-7):573–588.

AUTHOR BIOGRAPHIES

TILL KÖSTER is an research associate and PhD student in the modeling and simulation group at the University of Rostock. His research with the DFG ESCeMMO project focuses on the efficient simulation of cell-biological multi-level models. He holds Masters degrees in both Computer Science and Physics. His email is till.koester@uni-rostock.de.

PHILIPPE J. GIABBANELLI, Ph.D., is an Associate Professor in the Department of Computer Science & Software Engineering at Miami University (USA). His research interests include network science, machine learning, and simulation. He has directed projects in developing, verifying, and optimizing cellular automata for HIV. He currently serves as track chair for the 2020 Spring Simulation Conference, and program chair for the 2020 ACM SIGSIM Principles of Advanced Discrete Simulations (PADS) conference. His email address is giabbapj@miamioh.edu. His website is https://www.dachb.com.

ADELINDE M. UHRMACHER is head of the modeling and simulation group at the University of Rostock. Her research focuses on the development of modeling and simulation methods for multi-level systems and their application in areas such as cell biology, demography, or ecology. Methodological developments of her group include domain-specific languages for modeling and simulation, methods for automatically generating and executing simulation experiments, efficient simulation algorithms, and methods that assist in conducting and documenting simulation studies. She was editor in chief of SCS Simulation (2000-2006) and ACM Transactions of Modeling and Computer Simulation (2013-2019). Her e-mail is adelinde.uhramcher@uni-rostock.de

[D] GPU-Accelerated Simulation Ensembles of Stochastic Reaction Networks

Stochastic Simulation Algorithms are widely used for simulating reaction networks in cellular biology. Due to the stochastic nature of models and the large parameter spaces involved, many simulation runs are frequently needed. We approach the computational challenge by expanding the hardware used for execution by massively parallel graphical processing units (GPUs) to execute these ensembles of runs concurrently in a form of coarse-grained parallelization. Such computing infrastructure in the form of GPUs is readily available in desktop workstations and clusters but is not commonly exploited as part of stochastic simulation studies. Building on the existing literature in the field, we employ state-of-the-art algorithms to study the degree to which GPUs can augment the computation resources available for ensemble studies. Furthermore, the challenge of efficient work assignment given the GPU's synchronous mode of execution is explored. There are several algorithmic tradeoffs to consider for models with different execution characteristics, which we investigate in a performance study across four different models. To explore the limitations of the GPU-based simulators, the performance characteristics when executing large models are compared to those of highly optimized CPU simulators. Our results indicate that for some models adding a typical desktop GPU has a similar effect on performance as up to 40 added CPU cores.

[272] T. Köster, L. Herrmann, P. Andelfinger, and A. M. Uhrmacher, *GPU-Accelerated Simulation Ensembles of Stochastic Reaction Networks*, in *Winter Simulation Conference (WSC 2022)* (IEEE, 2022), pp. 2570–2581

Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds.

GPU-ACCELERATED SIMULATION ENSEMBLES OF STOCHASTIC REACTION NETWORKS

Till Köster
Leon Herrmann
Philipp Andelfinger
Adelinde Uhrmacher

Institute for Visual and Analytic Computing
University of Rostock
Albert-Einstein-Str. 22
Rostock, 18059, GERMANY

ABSTRACT

Stochastic Simulation Algorithms are widely used for simulating reaction networks in cellular biology. Due to the stochastic nature of models and the large parameter spaces involved, many simulation runs are frequently needed. We approach the computational challenge by expanding the hardware used for execution by massively parallel graphical processing units (GPUs) to execute these ensembles of runs concurrently in a form of coarse-grained parallelization. Such computing infrastructure in the form of GPUs is readily available in desktop workstations and clusters but is not commonly exploited as part of stochastic simulation studies. Building on the existing literature in the field, we employ state-of-the-art algorithms to study the degree to which GPUs can augment the computation resources available for ensemble studies. Furthermore, the challenge of efficient work assignment given the GPU's synchronous mode of execution is explored. There are several algorithmic tradeoffs to consider for models with different execution characteristics, which we investigate in a performance study across four different models. To explore the limitations of the GPU-based simulators, the performance characteristics when executing large models are compared to those of highly optimized CPU simulators. Our results indicate that for some models adding a typical desktop GPU has a similar effect on performance as up to 40 added CPU cores.

1 INTRODUCTION

Alongside traditional wet-lab experiments, simulation has evolved to be a core branch of scientific work in the domain of cellular biology. One particularly successful class of models is that of (stochastic) reaction networks (Loskot et al. 2019). For this model type, the system's state consists of specific amounts of various species or reactants. Only the total amount of each species is considered, not their spatial distribution as an approximation of the *well-stirred system* (Gillespie 2007). The amounts are propagated through time using a set of reactions. Each reaction updates the state vector by decreasing the amount of some species and increasing the number of others. The likelihood of a reaction firing (or, more generally, the speed of the reaction) is determined by the propensity. This propensity is a function of the current state of the system. The most common way to calculate the propensity is through *mass action kinetics* where the propensity is proportional to the product of the amount of the individual reactants multiplied with a reaction constant. Reaction networks can be executed deterministically or stochastically. In the latter case, which we will focus on for this work, instead of mapping the reaction network into a set of coupled differential equations, a *stochastic simulation algorithm* (SSA) executes individual reactions stochastically as discrete events in continuous time, and updates time and state vector accordingly. Thereby, most of the SSAs (as does our approach) assume the time intervals between successive events to be exponentially distributed.

```
step():
  k = select_weighted_random(propensities)
  execute_reaction(k)
  for i in all_reactions:
    propensity[i] = calc_propensity(i)
```

(a) Direct method

```
step():
  k = select_weighted_random(propensities)
  execute_reaction(k)
  for i in dependent_reactions(k):
    propensity[i] = calc_propensity(i)
```

(b) Optimized method

Figure 1: Pseudocode excerpt for a simulation step in two stochastic simulation algorithms.

Stochastic simulations are computationally expensive, and due to their stochasticity require multiple replications. Most simulation experiments also rely on the variation of parameter configurations, e.g., simulation-based optimization, sensitivity analysis, or general parameter scanning (Kleijnen et al. 2005), increasing the required number of computations further. In this work, we will focus on the problem of replications, but the concepts introduced are also suitable for parameter studies.

The computationally challenging portion of SSA consists of two parts: randomly selecting a reaction to fire, weighted by its computed propensity, and updating the state and its propensities. The Direct Method (DM) (Gillespie 1977) (and its equivalent formulation, the First Reaction Method (FRM) (Gillespie 1976)) compute all propensities for all reactions at every step. A far more efficient approach for larger systems is to use a dependency graph to only update those propensities that could have changed. The difference in pseudocode of the two approaches is shown in Fig. 1. This dependency-based approach is used in the Optimized Direct Method (ODM) (Cao et al. 2004) for the DM and in the Next Reaction Method (NRM) (Gibson and Bruck 2000) for the FRM.

In addition to these algorithmic improvements, parallelization is another option to increase simulation performance (Fujimoto 2016). However, the steps in SSA are often tightly coupled as each executed reaction may change the propensities of many other possible reactions. Since each step is relatively fast (less than a microsecond), efficient fine-grained parallelization (i.e., of a single replication) is challenging (Dematté and Prandi 2010). As many replications have to be executed, a coarse-grained parallelization across large numbers of replications (also called ensembles), provides another viable option to increase the performance of SSA simulations, also on the GPU (Li and Petzold 2010).

Code generation is an established technique to generate an executable with minimal overhead from generic functionality (Futamura 1999). By providing the compiler with both the simulation algorithm and the specific model, compiler optimizations can consider the simulation as a whole. Code generation works particularly well in the context of SSA, where a large part of the computational load lies in computing model-specific (propensity) expressions (Köster et al. 2022).

Our aim is to study the extent to which GPU implementations following state-of-the-art principles can augment the computational power provided by traditional CPU-based implementations in large ensemble studies. To this end, we present detailed performance measurements of three variants of SSA on GPUs across models with different execution characteristics and scales. Our simulator implementations, data and experiment scripts are publicly available (https://git.informatik.uni-rostock.de/mosi/gpu-ssa-ensemble).

2 BACKGROUND AND RELATED WORK

In the following, we introduce the hardware characteristics of GPUs as well as the existing work on GPU-based acceleration of ensembles of SSA simulations.

2.1 General-Purpose Computing on GPUs

GPUs have evolved from fixed-function hardware targeting the rendering of three-dimensional scenes to general-purpose accelerators widely available in machines ranging from laptops to supercomputers. Common GPU frameworks such as OpenCL (Khronos OpenCL Working Group, Beaverton, OR, USA

2011) and NVIDIA CUDA (Cook 2012) allow developers to specify GPU programs, referred to as *kernels*, on the level of individual GPU *threads*. Threads are grouped into sets of a hardware-specific size of 32 or 64 threads referred to as warps or wavefronts. Within a warp, threads operate in lockstep. Hence, if the control flow of the threads within a warp diverges, all branches are taken in sequence while discarding unneeded results. In addition, if adjacent threads access adjacent locations in memory, the accesses are *coalesced* into a single memory transaction served as one.

Due to the architectural focus on executing thousands of arithmetic operations in parallel rather than on control flow and caches, algorithms involving regular control flow and memory access patterns are a natural fit for GPU-based parallelization. Although simulations often involve sparse and irregular computations and thus require GPU-specific adaptations and optimizations, simulations from various domains, including cell biology, have been shown to benefit immensely from GPU-based parallelization given suitable choices of algorithms and data structures (Park and Fishwick 2010; Zhu et al. 2011; Xiao et al. 2019).

2.2 SSA on GPUs

The parallelization strategies for GPU-accelerated SSA simulations found in the literature can be divided into two groups. In coarse-grained parallelization, multiple *entire simulations* are executed in parallel. Fine-grained parallelization, on the other hand, also parallelizes *individual simulations*, in addition to executing multiple simulations concurrently, for instance in (Sumiyoshi et al. 2015). In the Direct Method, for example, a fine-grained approach may parallelize the updates of the population counts and propensities. The choice between coarse-grained or fine-grained parallelization determines the amount of GPU resources that can be dedicated to each individual simulation run. Fine-grained approaches can utilize many parallel threads to execute a single run. Hence, coarse-grained parallelization, which is the prevalent method in the literature, focuses on supporting particularly large numbers of replications of comparatively small model instances.

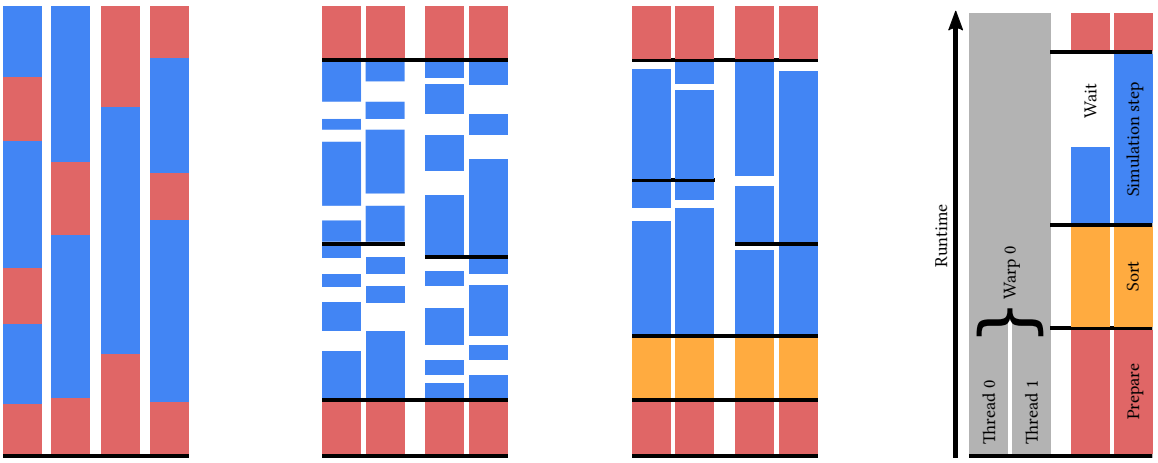
The Direct Method is used in these implementations (Jenkins and Peterson 2010; Klingbeil et al. 2012; Sumiyoshi et al. 2015), as its relatively simple control flow makes it reasonably straightforward to map to GPU execution. The simulator in (Li and Petzold 2010) uses the Logarithmic Direct Method. Dependency graph-based methods perform better for larger models, but are more difficult to implement and less prevalent in the literature. In (Klingbeil et al. 2011), the Next Reaction Method and the Logarithmic Direct Method are implemented in addition to the Direct Method. The closest existing work to ours presents an implementation of the Optimized Direct Method (Komarov and D'Souza 2012). To enable additional fine-grained parallelization, propensity values and their respective sums are updated in parallel using a prefix sum algorithm.

GPUs achieve highest performance when executing programs with largely homogeneous control flow and memory access patterns, which runs counter to the sparsity and heterogeneity of the computations involved when executing various parametrizations and replications of stochastic simulations in parallel. Thus, GPU-based SSA implementations require careful consideration of the GPU's execution model and hardware properties. A frequently employed means of optimizing execution speed is the efficient usage of the GPU's memory hierarchy. For instance, (Klingbeil et al. 2012) analyzes different data structures to store model parametrizations in fast read-only regions of GPU memory. Consideration is also given to dense and compressed storage of data (Komarov and D'Souza 2012). In our work, we generate specialized simulators using partial evaluation via code generation (Futamura 1999), in which full knowledge of the considered model is available during compilation, allowing the compiler to generate optimized model-specific machine code. We previously demonstrated the benefits of code generation for CPU-based SSA (Köster et al. 2022). In the GPU context, code generation avoids the need for individual threads to traverse complex data structures at simulation runtime and thus reduces the need for manual optimization of memory access patterns.

The issue of thread divergence can be addressed by sorting the computations involved in different simulation replications so that adjacent threads execute the same control flow (Kunz et al. 2012). We

present detailed measurements assessing the performance benefits and overhead when sorting impending reactions of SSA replications.

3 OUR APPROACH



(a) CPU (b) GPU, no sorting (c) GPU, sorting (d) Legend

Figure 2: An illustration of the methods we use for efficient execution. On the CPU, different threads do not interact and run independently (2a). When running on the GPU, different simulation steps of the different replications in the ensemble diverge, leading to serialization (2b). An added initial sorting step reduces divergence during execution (2c).

This section will give an overview of our approach to executing ensembles of SSA runs to GPU execution, noting where we differ from and expand the state of the art. In contrast to most of the related work, our approach uses the Optimized Direct Method (ODM) (Cao et al. 2004) in conjunction with code generation. The ODM has known good scaling behavior. The used data structures are simpler than in the NRM and thus lend themselves more to mapping to GPUs.

We considered three different approaches, all of which use coarse-grained parallelization, i.e., parallelization across multiple simulations. In preliminary experiments with a fine-grained parallelization, we observed no performance benefits for the models considered in this paper.

3.1 Code Generation

Traditionally, SSA simulators adhere to a strict separation between the simulator and the model comprised of production rules and initial populations. In this approach, generic simulator procedures handle functionalities such as the updating of propensities during a reaction, with dynamic concrete control flow and memory accesses according to the model data. We have previously shown that the performance of CPU-based SSA simulations can benefit immensely from code generation (Köster et al. 2020), which produces specialized simulation code combining a simulator and a specific model. Code generation offers the opportunity for compiler optimizations based on full knowledge of the simulator and the model. For instance, knowledge of the dependencies among the reactions reveals the number of propensity updates per reaction type and the affected propensities. In the GPU context, static optimizations of the memory access patterns may increase the opportunities for memory access coalescing, at the potential cost of increased divergence given the separation of update procedures by reaction type.

3.2 Propensity Sorting

The selection of the next reaction to occur involves a weighted sampling from the list of propensities. By sorting the propensities based on values gathered from previous simulation runs, as originally described in the context of the ODM (Cao et al. 2004), more efficient sampling can be achieved. The goal of the propensity sorting is to speed up the linear reaction selection by moving frequently firing reaction to the beginning of the array to be searched. The sorting occurs only once as a preprocessing step. Initial simulation runs are executed to count how often each reaction fires in the course of a simulation. The propensity list is then sorted using these reaction counts as estimates for future simulations.

3.3 Computation Sorting

As a first approach, we implemented an ODM simulation step in a single kernel that updates the populations and propensities affected by a reaction. ODM employs a dependency graph to avoid unnecessary updates. Thus, in a given simulation step, the different replications may execute different reactions which causes GPU threads within a warp to diverge both in the execution and the update step (Fig. 1b). The resulting serialization within warps is illustrated in Figure 2b. In this first approach, we do not mitigate the effects of divergence.

This can be contrasted with the Direct Method, which recalculates all propensities (Fig. 1a) after every reaction and thus does not scale well to large reaction counts. However, since the Direct Method lends itself to straightforward implementation targeting GPUs, we will use it as a baseline in our experiments.

Our second approach aims to minimize divergence by sorting the replications by the reaction they choose in each simulation step. The basic simulation algorithm and its implementation remains unchanged. However, simulation replications are allocated to the threads in a sorted fashion according to the selected reactions. This increases the likelihood that identical reactions occur in adjacent threads, reducing divergence. The sorting introduces additional computational as well as memory access overhead. It should, however, decrease thread divergence, because most consecutive threads execute the same reactions, as long as the number of possible reactions in the system is significantly smaller than the number of threads times the number of replications. Figure 2c visualizes how the sorting takes additional execution time, but increases the effective parallelism. For contrast, we also show the execution on a CPU (cf. Fig. 2a): since CPU threads work largely independently of each other, the individual simulations can progress independently.

We implemented two different variants of sorting. In the *logical sorting* method, only an index map is sorted based on the next reaction indices. The simulation variables are then accessed according to the sorted map so that most consecutive threads execute the same reaction. The *physical sorting* method, on the other hand, rearranges all simulation state variables based on the chosen next reactions in memory. The main difference between these sorting methods lies in the resulting access patterns to the simulation state memory. Since only an index map is sorted within the logical sorting method, the memory accesses made by adjacent threads are still scattered to different portions of the simulation state, limiting the opportunities for memory access coalescing (cf. Section 2.2).

A more favorable memory access pattern is achieved by the physical sorting method, in which the n-th thread processes the simulation variables at the n-th index. On the other hand, the physical sorting method incurs a higher memory access overhead during the sorting step.

3.4 Implementation

Our GPU simulators are written in C++ and use NVIDIA CUDA together with the high-level library Thrust (Bell et al. 2017), which provides data parallel primitives such as scans, sorts and reductions. The simulator code and our plotting scripts are available at https://git.informatik.uni-rostock.de/mosi/gpu-ssa-ensemble.

We reused the model parsing source code from our previous work targeting CPUs (Köster et al. 2020) but created a new GPU compatible simulator template. Code generation comes with a higher one-time cost of compiling the generated code, but in our scenario of interest, this is of minor concern, as we are interested

Table 1: Overview of the evaluated models, together with their respective amounts of species and reactions. The model degree describes the average number of propensities (based on the dependency graph) that must be updated after executing a single model reaction.

Model	Species	Reactions	Degree
SIR with N regions	3N	~4N	~6
Decay dimerization	3	4	3
Multistate	9	18	7
Multisite	66	288	55

in many replications where these one-time costs amortize. Compilation times for small to medium-sized models were similar to a larger CUDA program (On the order of 10 to 100 seconds for the models tested).

The next reaction indices for each replication are chosen through linear weighted random sampling. An alternative would have been a binary search as described for the LDM (Li et al. 2006). We chose the linear search for our implementation because it was originally described for the ODM (Cao et al. 2004), and in our experience works well for many models of practical relevance when used in conjunction with propensity sorting. For comparability, we use the linear search mode in both the GPU and the CPU simulators. The ODM uses two random numbers in each simulation step for each replication. Those random numbers are generated using the CUDA's default XORWOW pseudo-random generator.

Our implementation deviates from the original descriptions from our CPU-based simulator (Köster et al. 2022) with respect to the updates of the sum of all propensities, which is required in each simulation step. While those updated the propensity sum incrementally based on the change in the current step, our preliminary experiments showed that in the GPU context, it is typically more efficient to sum all propensities in each step instead.

4 EXPERIMENTS

We consider four models to test the suitability of the different simulators and to obtain a comparison to the sequential implementation. Table 1 gives an overview of the models. To provide a comprehensive comparison to a CPU-based execution, we include model instances larger than typically encountered in the literature, where models with less than 10 reactions are prevalent.

The **SIR (Susceptible, Infected, Recovered) model with regions** is a variant of the classic SIR model (Kendall 1956). The regular SIR model describes the course of a disease in a closed population. Viewed as a reaction network, the model includes three species – Susceptible, Infected and Recovered – which can react in the following two ways:

$$S + I \to I + I \qquad I \to R \ .$$

The first reaction describes an infection and the second one a recovery. The SIR model with regions arranges multiple of these SIR models in a circle. An example of our SIR region model with four regions is visualized in Figure 3. The simulation is initialized with a single infected individual present in the overall system.

The **decay dimerization model** (Gillespie 2001) is based on the following four reactions that describe the possible interactions between three species:

$$S_1 \to \emptyset \qquad S_1 + S_1 \to S_2 \qquad S_2 \to S_1 + S_1 \qquad S_2 \to S_3 \ .$$

We parametrized the model using the reaction rates and initial populations from (Gillespie 2001). All simulations were executed up to a simulation time of 10. Simulation time here is the virtual time within the model, measured in the model's internal unit system.

The **Multistate and the Multisite model** (Blinov et al. 2004) represent typical biochemical networks. Both were designed for the illustration of biochemical reaction networks and deal with the phosphorylation

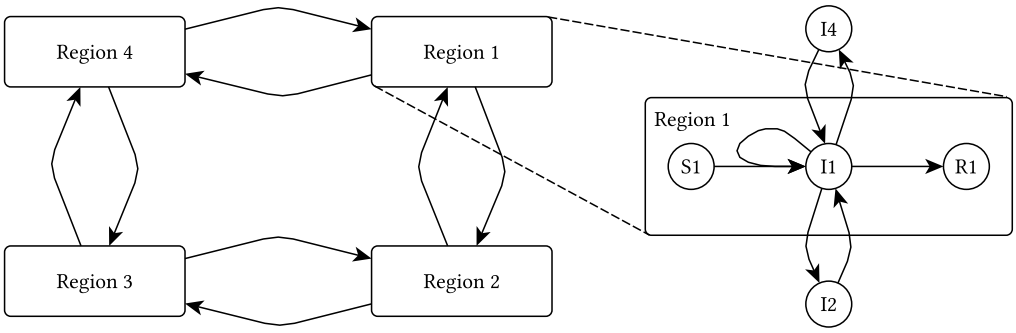


Figure 3: Structure of the SIR model with four regions.

as either different states of the molecules or at different sites of the molecules. They were simulated up to a time of 10 and 4 respectively. The models have been used previously in a performance review (Gupta and Mendes 2018) as well as for the evaluation of the CPU simulator (Köster et al. 2022) used as a baseline in our performance evaluation.

4.1 Reference Simulators and Environment

We compare the performance of our GPU implementations to our CPU-based simulator (Köster et al. 2020). This simulator implements a variant of the ODM and uses code generation in combination with partial evaluation to create specialized simulators for different stochastic models. We previously showed that for the Multistate and the Multisite model, this CPU-based simulator outperforms both the simulator integrated into BioNetGen as well as the simulators evaluated in a recent performance review (Gupta and Mendes 2018; Köster et al. 2022).

In addition to our GPU simulator variants based on ODM, we implemented a baseline GPU simulator using the original Direct Method without reaction sorting and code generation. While we expect the Direct Method's performance to be comparatively low for large models, this simulator variant provides an indication of the performance of a basic SSA implementation on a GPU. The experiments with this simulator serve to gain a better understanding on how the use of ODM, computation sorting, and code generation affects the performance. The simulations were run on a Windows 10 system with an i7-8700K Intel CPU with frequency scaling enabled, and an NVIDIA GeForce GTX 1070 GPU. The CPU simulator was evaluated running in Debian 10.5 on the Windows Subsystem for Linux (WSL2). This simulator was compiled using gcc version 8.3.0. The CPU simulator was executed using a single CPU thread. The C++ compiler version for the GPU simulators is 19.28.29912.0 (Visual Studio 16 2019) with CUDA version 10.2.

4.2 Results

In Figure 4, we give an overview of the different methods compared in our performance evaluation. The results of the performance evaluation can be found in Figure 5. As a performance metric, we use the wall-clock execution time in μs per step, which makes simulation results easily comparable across simulators and models. Five simulation runs consisting of many individual replications were executed to calculate the average performance values for the GPU simulators. The CPU performance is calculated as the average of ten measurements with 100 replications. All performance values are measured without code generation, compilation, or initialization times. The initialization time includes initial memory allocation as well as the transfer of initial simulation states to graphics memory. These costs amortize when running many replications. To better understand the impact of sorting, we measured the time taken to sort explicitly.

To verify correctness, we configured the GPU simulators to output the entire system trajectory, which involves substantial overhead. However, in real-world applications, the number of relevant observables

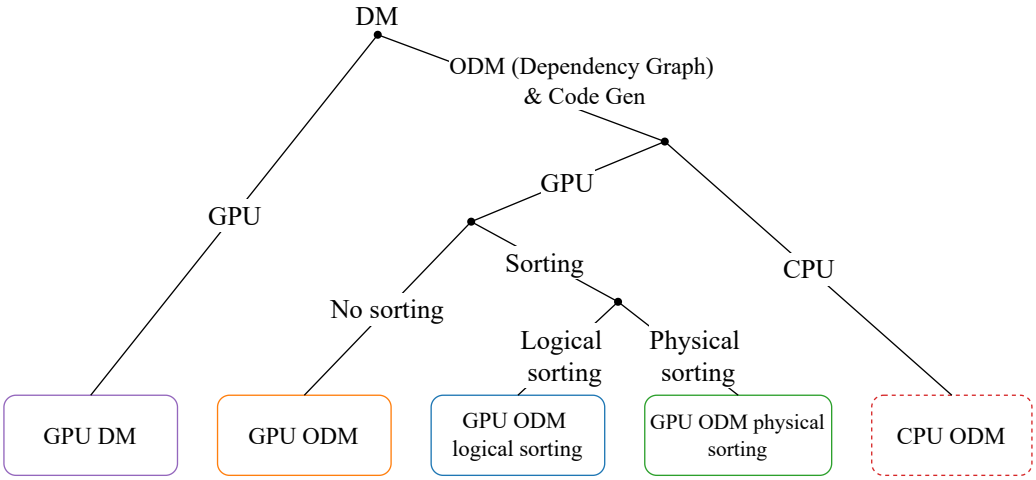


Figure 4: Hierarchy and relation of methods used for evaluation. Colors correspond to those used in Fig. 5.

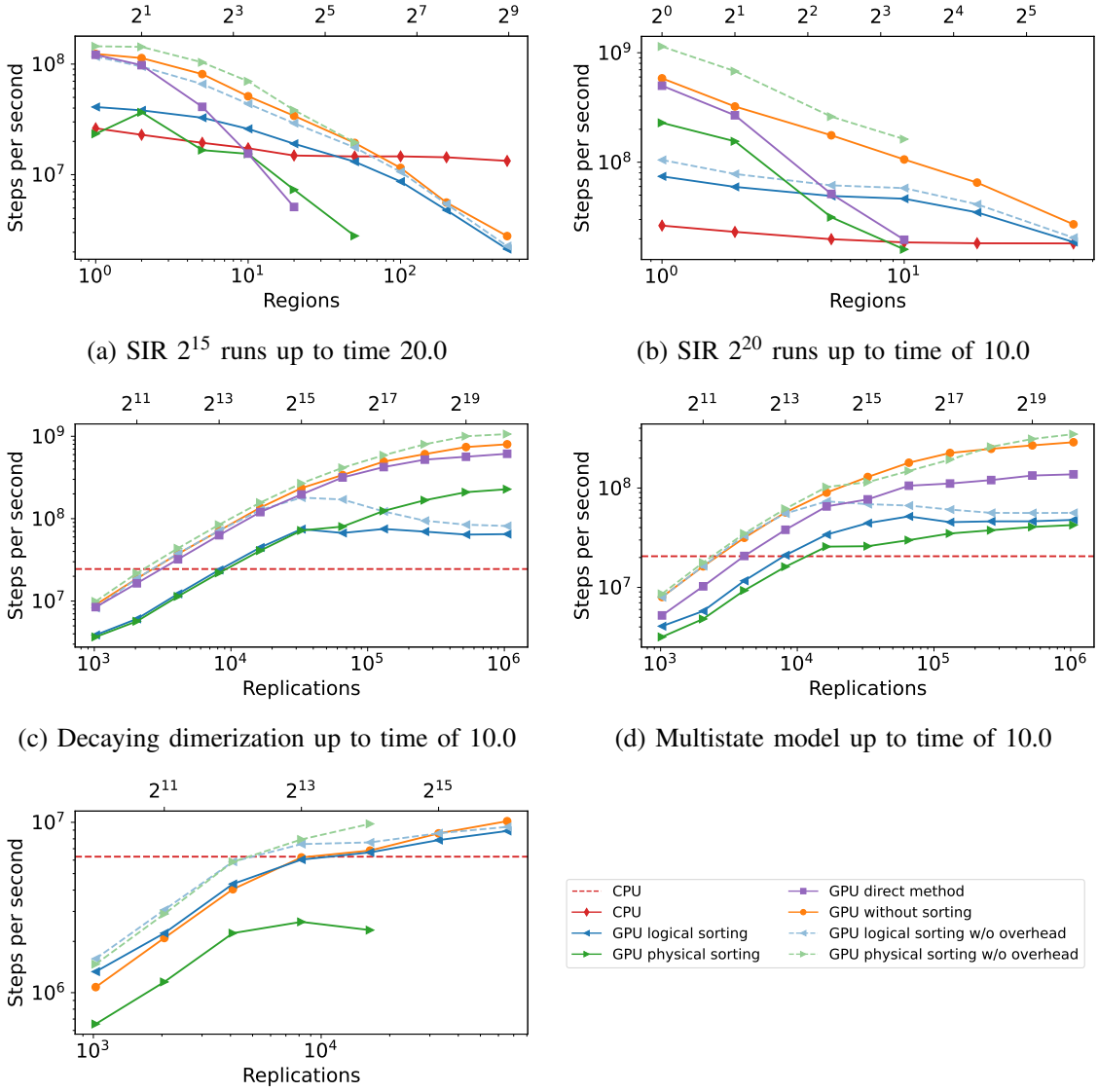
can be assumed to be small, introducing only little overhead. We therefore performed the measurements without output or storage of observations.

4.2.1 SIR Model with Regions

Figures 5a and 5b show the results for the performance measurements of two different model configurations of the SIR model with regions. The results shown in Fig. 5a originate from simulations with 2^{15} $(= 32\,768)$ replications and with a maximum simulation time of 20. Fig. 5b on the other hand, is based on simulations with 2^{20} $(\approx 10^6)$ replications which run up to a simulation time of 10. The results show that the performance of all simulators decreases with an increasing number of regions, which is to be expected as the reaction selection becomes more computationally expensive with larger models. Apart from the possibility of increased thread divergence due to larger numbers of reactions, the cost of the population and propensity updates should be unaffected by different sizes of the SIR region model, since each reaction updates two populations and at most six propensities.

However, in our GPU implementation, the cost of updating of the total propensity varies with different model sizes as all individual propensities are traversed. Compared to any of the GPU simulators, the performance of the sequential simulator is not as strongly affected by the varying model sizes. This can be seen both in Figures 5a and 5b. We attribute the more significant influence of the model size on the GPU simulators to the reaction selection and the recalculation of the total propensity. Linearly selecting the next reactions for all threads within a warp delays the progress of all threads until the slowest selection has completed. When the model instances grow in size and therefore have more possible reactions, it becomes more likely that one thread takes longer to choose a reaction and in turn, slows down all threads in the same warp. In addition, our implementation of the recalculation of the total propensities has to iterate over all propensities and consequently slows down for instances with more reactions.

Overall, the non-sorting method has the highest execution speed. The CPU simulator performs best only for large instances with relatively few replications. The GPU Direct Method performs well for small instances of the SIR region model, but slows down with larger instances, because it lacks the optimizations of the other simulators. Both sorting methods perform worse than the non-sorting method for this model. To understand the trade-off between the sorting overhead and its benefit during the execution of a simulation step, we also show results when subtracting the time spent on sorting. When disregarding the sorting overhead, the execution speed of the physical sorting method exceeds that of the non-sorting method. This shows that the additional sorting step has the potential to speed up the execution of the simulation steps. Still, the overhead introduced by this method far outweighs the performance gain.



(a) SIR 2^{15} runs up to time 20.0

(b) SIR 2^{20} runs up to time of 10.0

(c) Decaying dimerization up to time of 10.0

(d) Multistate model up to time of 10.0

(e) Multisite model up to time of 4.0

Figure 5: Performance graphs for different models and simulators.

The logical sorting method has a lower memory access overhead since only the index map is sorted. In contrast to the physical sorting method, subtracting the sorting overhead does not improve the performance significantly. For low numbers of replications (Figure 5a), the performance without the overhead is comparable to the non-sorting method. However, sorting only the indices when simulating many replications (Figure 5b) lowers the performance significantly. This is because rearranging the indices results in a non-optimal memory access pattern for the replication variables, which in turn provides fewer opportunities for memory access coalescing. This effect worsens with more replications.

4.2.2 Decay Dimerization Model

The Figure 5c shows that the execution speed of the decay dimerization model on a GPU increases with the number of replications. Overall, the execution speeds are higher than those of the SIR model, even though the regular SIR model with only a single region has only two reactions and a model degree of two,

compared to four decay dimerization reactions and a degree of three. This is because the SIR model may terminate after just one simulation step if the single initially infected entity heals before infecting other entities. The large variance in the simulation durations lowers the utilization of the GPU's resources. In contrast, the simulation durations of the decay dimerization model are more consistent, allowing for higher GPU utilization.

Similar to the SIR model results, the performance of the GPU simulator without sorting increases with the number of replications. Since the dimerization model is comparatively small and simple, the execution speed of the GPU Direct Method is roughly within 20% of the non-sorting ODM across all tested numbers of replications. Both sorting ODM simulators have similar execution characteristics up to about 2^{16} ($= 65\,536$) replications. Beyond that point, the performance of the physical sorting method increases further, whereas the speed of the logical sorting method stabilizes. When comparing the results to those of the SIR model, the performance of the physical sorting ODM is approximately equal or greater than that of the non-sorting ODM, but only when excluding the sorting overhead. The performance of the non-sorting GPU simulator, as well as the GPU Direct Method, scales almost linearly with the number of replications up to a certain number of replications, beyond which the experiment approaches full saturation of the GPU resources. The performance of the GPU simulator without sorting increases linearly up to about 2^{14} ($= 16\,384$) replications and saturates at a value of about 952 million simulation steps per second (1.05 ns per step). This throughput is the highest one of all simulators and models that were evaluated as part of this work. That means that the individual steps in this model are the cheapest to calculate compared to the other models. The sequential simulator reached an average execution time of about 24.5 million steps per second. The overall maximum speedup of the non-sorting GPU implementation is therefore about 39.

4.2.3 Multistate and Multisite Model

For the Multistate model, we find that the non-sorting method has, similar to the previous measurements, the highest overall execution speed when simulating large numbers of replications. Both sorting methods behave similarly when increasing the replication count. However, the sorting overhead of the physical sorting simulator is significantly higher than for the dimerization model because the Multistate model has more species, as well as more propensities to be sorted.

In general, the numbers of simulation steps per second are lower for all GPU simulators compared to the decay dimerization model. This can be attributed to the higher complexity, due to the larger number of reactions, and the higher numbers of species and reactions of the Multistate model. The Multistate model has 288 different reactions that update 55 propensities on average. We also observe an earlier diminishing of performance gains with more replications. The performance of the CPU simulator, on the other hand, is barely affected by the different structure of the Multistate model at 20.6 million steps per second compared to 24.5 million simulation steps per second for the dimerization model.

For the Multisite model, simulations were run up to a maximum simulation time of 4. The data shows that the implemented GPU methods are not well suited for this model. The performance of the CPU simulator running on two CPU cores would be better than that of all GPU simulators for all tested numbers of replications. One notable result, however, is that for this model, the logical sorting performs slightly better than the non-sorting approach for replication counts up to about 2^{12} ($= 4\,096$). The higher performance of the sorting approach can be attributed to the smaller computational and memory access overhead of the sorting step for fewer replications and its performance improvements due to minimizing branching for the propensity and population updates. Due to the large number of reactions and propensities to be updated in the Multisite model, the runtime overhead for the reaction selection and population and propensity updates caused by branching is higher than for the previously mentioned models. Measurements for the GPU Direct Method are not included in Fig. 5e, because of its poor performance for large and complex model instances.

5 CONCLUSION

This paper presented a detailed performance comparison of three variants of the Optimized Direct Method for simulating large ensembles of reaction networks on a GPU. A state-of-the-art CPU implementation and a basic GPU implementation served as baselines. The simulators were evaluated on the example of three models from the literature across a wide range of replication counts. Generally, we observed that full utilization of a modern GPU's computational resources is achieved only at large replication counts between 100 and 1000, depending on model size. In the best case, a GPU provided the equivalent of around 40 CPU cores executing an optimized CPU implementation.

In addition to the use of code generation, we introduced a sorting step to make a more homogeneous control flow across simulations. We find that for some problems, the sorting reduces the execution time of the simulation step substantially. However, this speedup does not outweigh the overhead introduced by the sorting. Future work could explore a localized sorting that would reduce control flow divergence while inducing lower overhead. A basic GPU implementation of the Direct Method performed well for very small model instances but did not scale with increasing instance size. An attractive avenue for future work is the exploration of improved memory layouts using thread-local storage, which exhibited promising results in our initial experiments.

ACKNOWLEDGEMENTS

Financial support was provided by the Deutsche Forschungsgemeinschaft (DFG) research grant ESCEMMO (UH-66/13).

REFERENCES

Bell, N., J. Hoberock, and C. Rodrigues. 2017. "THRUST: A Productivity-oriented Library for CUDA". In *Programming Massively Parallel Processors*, 475–491. Elsevier, Amsterdam.

Blinov, M. L., J. R. Faeder, B. Goldstein, and W. S. Hlavacek. 2004. "Bionetgen: Software for Rule-based Modeling of Signal Transduction Based on the Interactions of Molecular Domains". *Bioinformatics* 20(17):3289–3291.

Cao, Y., H. Li, and L. Petzold. 2004. "Efficient Formulation of the Stochastic Simulation Algorithm for Chemically Reacting Systems". *The Journal of Chemical Physics* 121(9):4059–4067.

Cook, S. 2012. *Cuda Programming: A Developer's Guide to Parallel Computing with Gpus*. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Dematté, L., and D. Prandi. 2010. "Gpu Computing for Systems Biology". *Briefings in bioinformatics* 11(3):323–333.

Fujimoto, R. M. 2016. "Research Challenges in Parallel and Distributed Simulation". *ACM Transactions on Modeling and Computer Simulation* 26(4):1–29.

Futamura, Y. 1999. "Partial Evaluation of Computation Process – an Approach to a Compiler-compiler". *Higher Order Symbolic Computation* 12(4):381–391.

Gibson, M. A., and J. Bruck. 2000. "Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels". *The Journal of Physical Chemistry A* 104(9):1876–1889.

Gillespie, D. T. 1976. "A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions". *Journal of Computational Physics* 22(4):403–434.

Gillespie, D. T. 1977. "Exact Stochastic Simulation of Coupled Chemical Reactions". *The Journal of Physical Chemistry* 81(25):2340–2361.

Gillespie, D. T. 2001. "Approximate Accelerated Stochastic Simulation of Chemically Reacting Systems". *The Journal of Chemical Physics* 115(4):1716–1733.

Gillespie, D. T. 2007. "Stochastic Simulation of Chemical Kinetics". *Annual Review of Physical Chemistry* 58(1):35–55.

Gupta, A., and P. Mendes. 2018. "An Overview of Network-Based and -Free Approaches for Stochastic Simulation of Biochemical Systems". *Computation*.

Jenkins, D., and G. Peterson. 2010. "GPU Accelerated Stochastic Simulation".

Kendall, D. G. 1956. "Deterministic and Stochastic Epidemics in Closed Populations". In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 4: Contributions to Biology and Problems of Health*, 149–165. University of California Press.

Khronos OpenCL Working Group, Beaverton, OR, USA 2011. *The Opencl Specification, Version 1.1*.

Kleijnen, J. P., S. M. Sanchez, T. W. Lucas, and T. M. Cioppa. 2005. "State-of-the-art Review: A User's Guide to the Brave New World of Designing Simulation Experiments". *INFORMS Journal on Computing* 17(3):263–289.

Klingbeil, G., R. Erban, M. Giles, and P. K. Maini. 2011. "STOCHSIMGPU: Parallel Stochastic Simulation for the Systems Biology Toolbox 2 for MATLAB". *Bioinformatics* 27(8):1170–1171.

Klingbeil, G., R. Erban, M. Giles, and P. K. Maini. 2012. "Fat Versus Thin Threading Approach on GPUs: Application to Stochastic Simulation of Chemical Reactions". *IEEE Transactions on Parallel and Distributed Systems* 23(2):280–287.

Komarov, I., and R. M. D'Souza. 2012. "Accelerating the Gillespie Exact Stochastic Simulation Algorithm Using Hybrid Parallel Execution on Graphics Processing Units". *PLOS ONE* 7(11).

Köster, T., T. Warnke, and A. M. Uhrmacher. 2022. "Generating Fast Specialized Simulators for Stochastic Reaction Networks Via Partial Evaluation". *ACM Transactions on Modeling and Computer Simulation* 32(2).

Kunz, G., D. Schemmel, J. Gross, and K. Wehrle. 2012. "Multi-level Parallelism for Time- and Cost-efficient Parallel Discrete Event Simulation on Gpus". In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, 23–32: IEEE.

Köster, T., T. Warnke, and A. M. Uhrmacher. 2020. "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*, SIGSIM-PADS '20, 159–170. New York, NY, USA: Association for Computing Machinery.

Li, H., and L. Petzold. 2010. "Efficient Parallelization of the Stochastic Simulation Algorithm for Chemically Reacting Systems on the Graphics Processing Unit". *The International Journal of High Performance Computing Applications* 24(2):107–116.

Li, H., L. Petzold, H. Li, and L. Petzold. 2006. "Logarithmic Direct Method for Discrete Stochastic Simulation of Chemically Reacting Systems". *Journal of Chemical Physics* 16:1–11.

Loskot, P., K. Atitey, and L. Mihaylova. 2019. "Comprehensive Review of Models and Methods for Inferences in Bio-chemical Reaction Networks". *Frontiers in Genetics* 10.

Park, H., and P. A. Fishwick. 2010. "A Gpu-based Application Framework Supporting Fast Discrete-event Simulation". *Simulation* 86(10):613–628.

Sumiyoshi, K., K. Hirata, N. Hiroi, and A. Funahashi. 2015. "Acceleration of Discrete Stochastic Biochemical Simulation Using GPGPU". *Frontiers in Physiology* 6.

Xiao, J., P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll. 2019. "A Survey on Agent-based Simulation Using Hardware Accelerators". *ACM Computing Surveys* 51(6):1–35.

Zhu, Y., B. Wang, and Y. Deng. 2011. "Massively Parallel Logic Simulation with Gpus". *ACM Transactions on Design Automation of Electronic Systems* 16(3):1–20.

AUTHOR BIOGRAPHIES

TILL KÖSTER is a doctoral researcher in the modeling and simulation group at the Institute for Visual and Analytic Computing, University of Rostock. His e-mail address is till.koester@uni-rostock.de.

LEON HERRMANN is a student in the Computer Science Master's program at the University of Rostock. His e-mail address is leon.herrmann@uni-rostock.de.

PHILIPP ANDELFINGER is a postdoctoral researcher in the modeling and simulation group at the Institute for Visual and Analytic Computing, University of Rostock. His e-mail address is philipp.andelfinger@uni-rostock.de.

ADELINDE M. UHRMACHER is a professor at the Institute for Visual and Analytic Computing, University of Rostock, and head of the modeling and simulation group. Her e-mail address is adelinde.uhrmacher@uni-rostock.de.

[E] Expressive modeling and fast simulation for dynamic compartments

Compartmentalization is vital for cell biological processes. The field of rule-based stochastic simulation has acknowledged this, and many tools and methods include provisions for compartmentalization. However, mostly, this is limited to a static compartmental hierarchy and does not integrate compartmental changes. Integrating compartmental dynamics is challenging for the design of the modeling language and the simulation engine. The language should support a concise yet flexible modeling of compartmental dynamics. As a basis of our work, we use a rule-based language for multi-level cell biological modeling called ML-Rules, which supports a wide variety of compartmental dynamics. We adapt its syntax slightly and develop an efficient simulation engine for compartmental dynamics. To execute a rule-based model that includes compartmental dynamics efficiently, we combine efficient data structures and new and existing algorithms and implement them in the Rust programming language. We evaluate our implementation using two case studies from existing cell-biological models. The execution of these models outperforms previous simulations of ML-Rules by two orders of magnitude. Finally, we present a prototype of a WebAssembly-based implementation to allow for a low barrier of entry when exploring the language and associated models without the need for local installation.

[273] T. Köster, P. Henning, T. Warnke, and A. Uhrmacher, *Expressive Rule-Based Modeling and Fast Simulation for Dynamic Compartments*, Plos One **19**, 312813 (2024)

RESEARCH ARTICLE

Expressive rule-based modeling and fast simulation for dynamic compartments

Till Köster[1], Philipp Henning[1,2]*, Tom Warnke[1,3], Adelinde Uhrmacher[1]

1 Institute for Visual and Analytic Computing, University of Rostock, Rostock, Germany, **2** Institute of Medical Biochemistry and Molecular Biology, University Medicine Rostock, Rostock, Germany, **3** Limbus Medical Technologies GmbH, Rostock, Germany

* philipp.henning@uni-rostock.de

Abstract

Compartmentalization is vital for cell biological processes. The field of rule-based stochastic simulation has acknowledged this, and many tools and methods have capabilities for compartmentalization. However, mostly, this is limited to a static compartmental hierarchy and does not integrate compartmental changes. Integrating compartmental dynamics is challenging for the design of the modeling language and the simulation engine. The language should support a concise yet flexible modeling of compartmental dynamics. Our work is based on ML-Rules, a rule-based language for multi-level cell biological modeling that supports a wide variety of compartmental dynamics, whose syntax we slightly adapt. To develop an efficient simulation engine for compartmental dynamics, we combine specific data structures and new and existing algorithms and implement them in the Rust programming language. We evaluate the concept and implementation using two case studies from existing cell-biological models. The execution of these models outperforms previous simulations of ML-Rules by two orders of magnitude. Finally, we present a prototype of a WebAssembly-based implementation to allow for a low barrier of entry when exploring the language and associated models without the need for local installation.

OPEN ACCESS

Citation: Köster T, Henning P, Warnke T, Uhrmacher A (2024) Expressive rule-based modeling and fast simulation for dynamic compartments. PLoS ONE 19(10): e0312813. https://doi.org/10.1371/journal.pone.0312813

Editor: Claudio Zandron, University of Milano–Bicocca: Universita degli Studi di Milano-Bicocca, ITALY

Received: June 26, 2024

Accepted: October 14, 2024

Published: October 31, 2024

Peer Review History: PLOS recognizes the benefits of transparency in the peer review process; therefore, we enable the publication of all of the content of peer review and author responses alongside final, published articles. The editorial history of this article is available here: https://doi.org/10.1371/journal.pone.0312813

Data Availability Statement: All codes used in this study is available at https://git.informatik.uni-rostock.de/mosi/ml-rules3-official.

1 Introduction

Compartmentalization is a central aspect of cell biological systems [1]. Modeling such systems as interrelated compartments in and between which processes occur allows approximating the system structure and its effect on the processes. To include the causal influence of compartments on the processes, it suffices to support *static compartments*, that is, compartments that do not change. However, to account for the causal influence of the occurring processes on the compartmental structure and interactions, the model must support *dynamic compartments*. This includes the creation of new compartments, their deletion, merging, and splitting of compartments, as well as one compartment entering or leaving another (Fig 1).

The reciprocal influence of compartments and the processes that occur in and between them can be observed in many cell biological systems. For example, the life cycle of a cell can be modeled as intracellular processes affecting cell growth, ultimately leading to cell division

Funding: P.H. received funding from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation, https://www.dfg.de/en) SFB 1270 – 299150580 ELAINE and the Young Neuro Scientist Programme of the Centre for Transdisciplinary Neurosciences Rostock (CTNR). T.K. received funding from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation, https://www.dfg.de/en) grant 225222086. The DFG or CTNR did not play a role in the study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing interests: The authors have declared that no competing interests exist.

[2]. Intercellular communication in cell populations affects the intracellular behavior of individual cells. Signaling pathways include intracellular compartmental processes such as endocytosis or the transfection of lipoplexes. Although such processes can sometimes be approximated with static compartments, dynamic compartments are the natural method for precise and expressive models.

Several modeling approaches that support dynamic compartments have been proposed in the past [3–6]. However, in contrast to approaches limited to static compartments, running simulations for these modeling approaches proved too computationally challenging for usage in relevant biological applications. In fact, some of the approaches developed were never equipped with a (publicly available) simulator implementation.

In the following, we present an approach that overcomes these performance problems and makes dynamic compartments feasible for studying cellular processes. In particular, our contributions are:

- an analysis of rule-based language design choices on efficiently executing dynamic compartments,
- an analysis of execution requirements for an expressive rule-based language, such as ML-Rules, which allows compact and expressive specifications of models with dynamic compartments [7],
- a design of a new efficient stochastic simulation algorithm tailored to models with dynamic compartments combining various algorithms with code generation,
- an implementation of a new modeling and simulation tool ML-Rules 3 in Rust based on the new efficient stochastic simulation algorithm, and slight adaptations in comparison to earlier implementations of ML-Rules; the latter includes syntactic enhancements to improve usability, like type checking of units of measurements, and a new type of attributes to facilitate efficient simulation,
- a realization of a WebAssembly-based web simulation tool at http://mlrules.pages.dev to enable simple deployment of the tool,
- and an evaluation of the expressiveness and performance of ML-Rules 3 based on case studies from cell biology, including an mRNA delivery model, where based on dynamic compartments, due to ML-Rules 3, now closer matches to biological results become possible.

1.1 Compartmental dynamics

Cell biology considers compartments essential elements of cell behavior [8]. Compartments affect cellular processes by controlling both the reactions that can occur and the rate at which they do. They are considered important to modeling signaling pathways, in which extracellular ligands trigger a cascade of biochemical reactions that span various cellular compartments, such as membrane, cytosol, and nucleus [9]. A prominent example is the canonical Wnt/β-catenin signaling pathway, which is essential for cellular functions such as proliferation and differentiation and is involved in several diseases, including cancer [10]. In the last 20 years, more than 20 quantitative models have been built to analyze the Wnt/β-catenin signaling pathway [11]. Many of these models express the causal influence of compartments on processes, e.g., constrain processes to specific compartments or study proteins shuttling between compartments [12, 13]. Also, the volume of the compartment may influence the reactants' density and, consequently, the kinetics within the different compartments [9, 14]. Compartmental

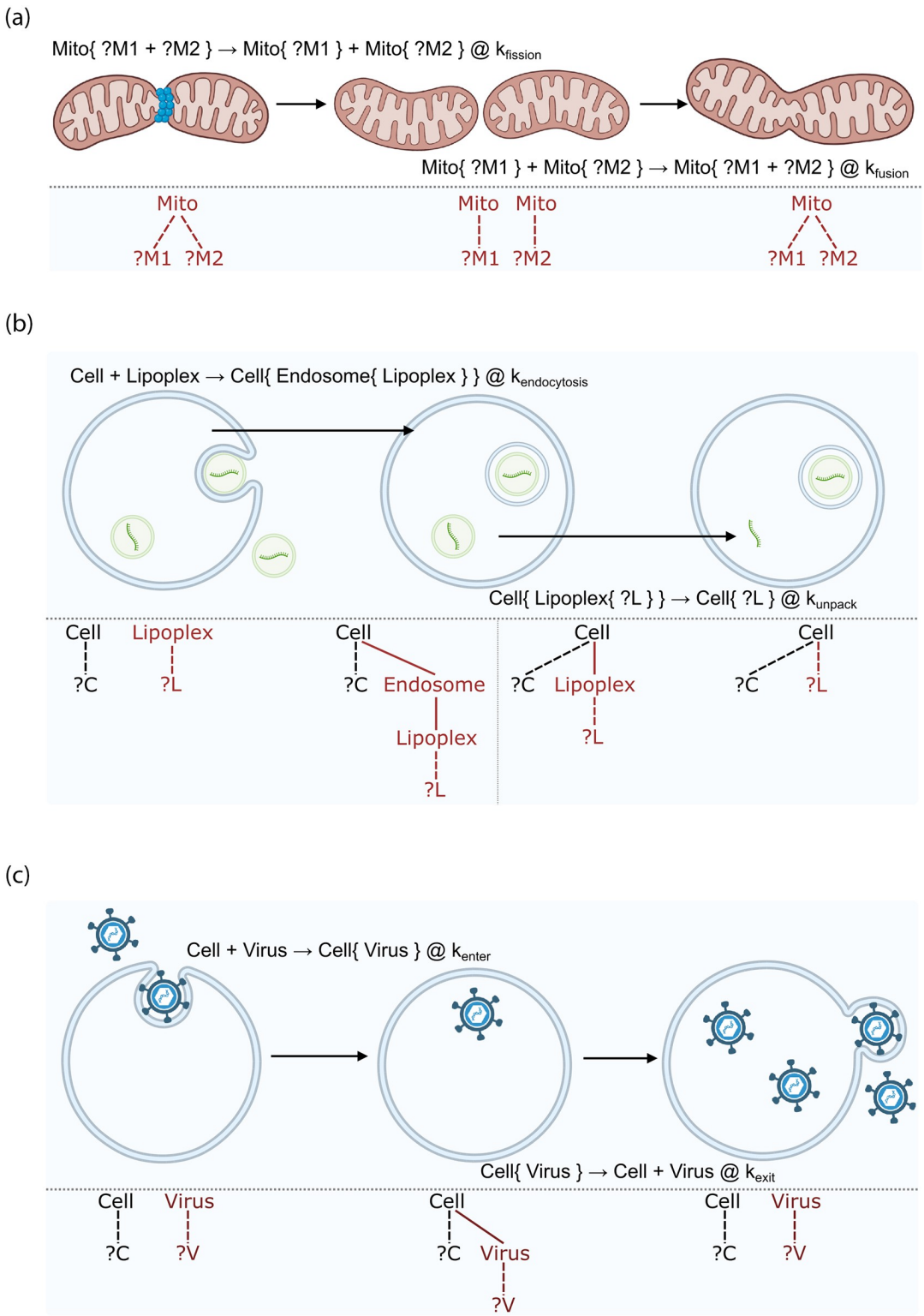


Fig 1. Examples of compartmental dynamics in cell biology. The figure shows a sketch of the processes, the specification in ML-Rules 3, and the n-ary tree structure of the term rewriting that underlies ML-Rules interpretation. The red lines in the n-ary tree structure indicate structures that are changed during the reaction. For the rules, we use the ML-Rules notation. The left side is transformed (using the →) into the right side at a rate (following the @ symbol). The {} denote nesting relationships. (**a**) Fission and fusion of compartments: A large mitochondrium can divide into two smaller ones, whereby its content (here ?M1

and ?M2) is distributed between the two new mitochondria. Two mitochondria can also fuse into a large one containing the solution of both smaller ones. (**b**) Creation and removal of compartments: During the endocytosis of the lipoplex, an endosome is formed around it. Inside the cell, the lipoplex can unpack its content (?L) into the cell. (**c**) Shuttling of compartments: A virus carrying DNA or RNA can enter a cell. New viruses produced in the cell can level it and spread.

https://doi.org/10.1371/journal.pone.0312813.g001

constraints on reactions are supported by most simulation tools and standardized modeling exchange formats such as SBML [14].

Considering dynamic compartments opens up new possibilities. Treating compartments as regular species with attributes enables causal effects between compartments and of the compartmentalization on underlying processes, called *downward causation* [6]. Causality also occurs in the opposite direction— intra-compartmental dynamics can influence the compartmental level, including its attributes such as volume. This *upward causation* can also result in changes to the compartmental structure, such as fission and fusion, or the creation and deletion of compartments.

The changes in the compartmental hierarchy occur during the simulation execution. Therefore, they have been called *dynamic compartments* [15]. Dynamic compartments are one form of variable structure model [16] and might occur at and involve various levels of cellular organization [6]. Signaling interactions within multi-cellular populations synchronize and organize the individual cells' behavior in dynamic processes such as tumor growth [17] or morphogenesis [18]. Cell proliferation, migration, and differentiation are regulated by several signaling pathways, which again react to changes in the cells' environment. Dynamic compartments are also necessary for intracellular dynamics; during the last decades, it became evident that a static view of intracellular compartments does not suffice [8]. The internalization of receptors is crucial in regulating signaling pathways [19]. Receptors are sorted into different endosomal compartments when entering the endocytic pathway. To accomplish this sorting, the organelles undergo frequent compartmental dynamics such as maturation, transformation, fusion, fission, and degradation [20]. More recently, the development of mRNA vaccinations has increased the interest in studying liposomal dynamics [21] and its modeling [22] to provide effective drug delivery systems.

Compartmental dynamics take on various forms, as also discussed in [23] and modeling approaches such as Bioambients [3], Brane Calculi [24], or BetaBinders [25], including a) the fission and fusion processes of compartments, as in the case of mitochondria [26] (Fig 1a), b) the formation and disintegration of cellular compartments, as in the case of liposomal dynamics [27] (Fig 1b), and c) a compartment entering or leaving another compartment, as in the case of a viral entry or release [28] (Fig 1c).

1.2 Stochastic simulation

To simulate these compartmental dynamics, we will interpret compartments as discrete entities. We assume that the discrete compartments can be arbitrarily nested, that entities can exist within and outside of compartments, that compartmental dynamics rely on the explicit definition of reactions, and that the reactants are well-mixed within and outside of each compartment.

Based on these assumptions, adopting stochastic simulation algorithms (SSA), popularized under the term of Gillespie algorithms [29], appears most appropriate. These algorithms can be formalized using Continuous-Time Markov Chains (CTMCs) [30]. In CTMCs, the distributions for the waiting time until the next state transition and the successor state only depend on the current state. The sojourn time in a state is exponentially distributed, similar to a

physical decay process. The propensity is the expected rate of firing for a particular transition, given a specific model state. Each potential transition i between states in the CTMC has a propensity p_i.

In the *direct method* family of methods [31], a timestep is sampled using the exponential distribution. The total propensity sum $\sum_i p_i$ is used as the rate parameter of the exponential distribution. The reaction is selected by weighted random choice ($\mathcal{P}(i)\ p_i$). The *first reaction* family of methods [29] computes a time for every possible reaction by sampling from the exponential distribution (with the respective p_is) and then selecting the smallest one. Both original approaches are wasteful, as many computations might be needlessly redone. Major performance benefits result from storing results, such as propensities [32] or the time of the next event of a reaction [33], and updating this information on demand. For this update on demand, a dependency graph stores the dependencies between reactions and the associated propensities. The next reaction method [33] builds on the first reaction method and maintains a schedule of reactions and the time they will occur. After a reaction is executed, based on the dependency graph, the affected reactions are rescheduled. Another approach builds on the direct method, stores the propensities of reactions, and, based on the dependency graph, updates the affected propensities in each step [32].

1.3 Rule-based modeling and compartments

The modeler enumerates all possible species and reactions in species reaction models. This modeling approach is limited when dealing with complex systems, like proteins involving multiple binding sites, as the number of possible species and reactions can grow exponentially in these cases [34]. The fundamental idea of rule-based modeling (combined with CTMC semantics) is to use rewriting rules to specify transition classes of a CTMC. The left side of a rewriting rule specifies the reactants and their context as a pattern matched to the current state. Using patterns, a single rule can express a large set of reactions, which can be parametrized with the variables matched in the pattern. The patterns on the left rule side constrain the reactants participating in a reaction, for example, a pattern like $A(x) + B(x) \rightarrow \ldots$ expresses that one entity A and one entity B can only react if they share an attribute value x. Each successful match contributes one transition within the CTMC. The rewriting rule is annotated with a rate expression, which is evaluated based on the left side's match. This can include factoring mass action kinetics into the transition rate (based on the multiplicities of the reactants) as well as employing attribute values of the reactants or their context. The successor state of the resulting transition is computed by applying the rewriting operation: removing the reactants and adding the products.

Including dynamic compartments poses a challenge to rule-based modeling languages. Languages like Kappa [35, 36] or BNGL (as part of BioNetGen) [37, 38], for example, are based on graph rewriting rules. Their focus is on modeling interacting entities between which bonds are created and destroyed. Bonds are defined for exactly two entities, which precludes using them to express compartments. Static compartments can be integrated [9, 39]. They are used to restrict the scope within which a reaction takes place. Shuttling of simple (or bound) molecules between static compartments as they are commonly supported also by the modeling standards [14], can be easily expressed, e.g., by *Protein@Cytosol* → *Protein@Nucleus* in BioNetGen. However, here, neither the cytosol nor the nucleus forms a species and, as such, a potential reactant or product of a rule or reaction.

Dynamic compartments can be expressed using graph rewriting when extending the formalism with hyperedges. Hyperedges are those that can connect to more than two vertices. This approach is used in the modeling formalism React(C) [4]. Here, a compartment can be

denoted by a hyperedge, which can join any number of entities. In addition, the compartment itself can be represented as a species. The following rule describes a cell containing a nucleus and a protein that shuttles into the nucleus (Fig 1c):

$$Cell(c) + Nucleus(n, c) + Protein(c) \rightarrow Cell(c) + Nucleus(n, c) + Protein(n) \quad .$$

Note that the containment relation is expressed purely via variables c and n, identifying the cell and nucleus as a compartment (hyperedge). So the values of the variables c and n are identifiers for the respective compartments and as such unique. Entities not affected by the rule can be omitted on both rule sides, Therefore, we could have omitted $Cell(c)$. However, this would have made it harder to understand the role of the value of variable c.

In the approaches based on hypergraphs, new compartments can be created by introducing new, unused ("fresh") values for variables on the right side. Similar solutions to model dynamic compartments can be found in process algebras, e.g., the attributed PI calculus [40]. The following rule expresses the fission of an organelle, with the ν operator yielding fresh values (identifiers) for the two new cells, i.e., c_1 and c_2:

$$Cell(c) \rightarrow (\nu c_1, c_2) Cell(c_1) + Cell(c_2)$$

However, this cell fission example also shows a fundamental problem with the hypergraph approach: the effects on the contained entities can not be modeled as easily as the effect on the compartments. In the above reaction, all entities referencing the value of c as their containing compartment need to be updated to either c_1 or c_2. Also the fusion of compartments

$$Cell(c_1) + Cell(c_2) \rightarrow (\nu c) Cell(c)$$

would require that all entities referencing the values c_1 and c_2 would now need to reference c. These changes can be expressed with workarounds, for example, by defining intermediate states and additional rules for each species contained within the original compartment, with infinite propensities, but this clutters the model description and is computationally rather expensive. An additional problem with the extension of graph rewriting to hypergraphs is that finding occurrences of the left rule side in the current state is harder. Whereas graph rewriting can exploit some properties (e.g., Kappa exploits "rigidity" [41]) to speed up this process, similar optimizations for hypergraphs are not known. As a consequence, no simulation tool based on hypergraph rewriting has been published. In particular, no simulator for React(C) is available.

An alternative approach to graph rewriting is multiset rewriting, a special case of term rewriting [42, 43]. Here, the containment in compartments is structurally equivialent to an n-ary trees (see also Fig 1). Therefore, they can be expressed as terms with a variadic function symbol or, equivalently, an associative and commutative function symbol for forming multisets [44, 45]. The contents of a compartmental entity can then be represented by a multiset, usually denoted with the symbol + or, in infix notation. The protein movement rule above (Fig 1c) can be written as

$$Cell(Nucleus(?n) + Protein + ?c) \rightarrow Cell(Nucleus(Protein + ?n) + ?c) \quad ,$$

where $?c$ captures the remaining multiset within the cell after finding a nucleus and a protein in a cell, and $?n$ captures the contents of the nucleus. These variables starting with ? (called "sequence variables" in the rewriting literature [46]) play a central role, as they denote the entities unaffected by a rule and allow operations on those multisets of entities, i.e., the content of the compartments [47]. One line of work that very closely follows the idea of multiset rewriting

is Colored Stochastic Multilevel Multiset Rewriting (CSMMR) [5, 48]; another is ML-Rules [6, 49].

In multi-set rewriting, the cell fission rule could be written as

$$C(?c) \rightarrow C(?c_1) + C(?c_2) \textbf{ where } (?c_1, ?c_2) = splitHalf(?c) \ ,$$

binding the result of the function *splitHalf* to a pair of variables in a **where** block [47]. In the syntax of CSMMR, the rule would read as

$$C(x) \rightarrow let(z, y) = x \ in \ C(z) + C(y) \ .$$

In the syntax of ML-Rules 3, the operator "+" is overloaded to realize an inline function on cell content, which randomly assigns each element of C to bind either to the sequence variable $?c_1$ or $?c_2$, i.e.,

$$C(?c_1 + ?c_2) \rightarrow C(?c_1) + C(?c_2)$$

(see Fig 1a). In some biological processes, fission events occur asymmetrically. For example, such asymmetrical fission events are observed in mitochondria. The small mitochondria might include many damaged parts and can be removed from the cell, thereby contributing to the health of the mitochondrial network. To model this process weights can be assigned to the sequence variables

$$Mito(?m_1[R_1] + ?m_2[R_2]) \rightarrow Mito(?m_1) + Mito(?m_2)$$

where $R_1/(R_1 + R_2)$ and $R_2/(R_1 + R_2)$ are the ratios of elements assigned to $?m_1$ and $?m_2$, respectively.

While multi-set rewriting provides an expressive and readable way to model dynamic compartments (see Fig 1), languages that are built on multi-set rewriting, such as ML-Rules, are still hard to execute efficiently precisely because of their expressive power. One approach to alleviate this is to provide specialized simulators that exploit that some language features are not used in a given model [49], thus working on a subclass of models. Another subclass is considered in the modeling approach presented in [15]: it only considers models where compartments are not nested, and entities do not exist outside compartments.

2 Simulation engine

We implemented version 3 of ML-Rules using the Rust programming language to develop and test an efficient simulator for dynamic compartments. The model is specified within an external domain-specific language which is parsed into Rust code. The structure of the implementation and its components is shown in Fig 2. We have tuned the implementation for performance on the main expected code paths. That means we specifically applied several optimizations to the typical main loop at the potential cost of the unusual behavior. Overall optimizations include minimizing allocations, a flat, indexed-based data layout, and partial evaluation of repeated computations. The following sections will discuss differences from the previous versions and their implications on runtime performance and user experience.

Compared to the previous implementation, we introduced a few changes to the syntax of ML-Rules (Section 2.1). However, the focus of our research has been to develop a simulation approach that is able to handle models that may exhibit a wide range of compartmental dynamics efficiently. It incorporates a new variant of an SSA specifically designed for large SSA systems (Section 2.3). This variant of the SSA (labeled flat model simulator in Fig 2) is not specific to compartmental models, as compartmentalization is abstracted away in a previous step. One significant optimization is code generation for repeated expression evaluation 2.4.

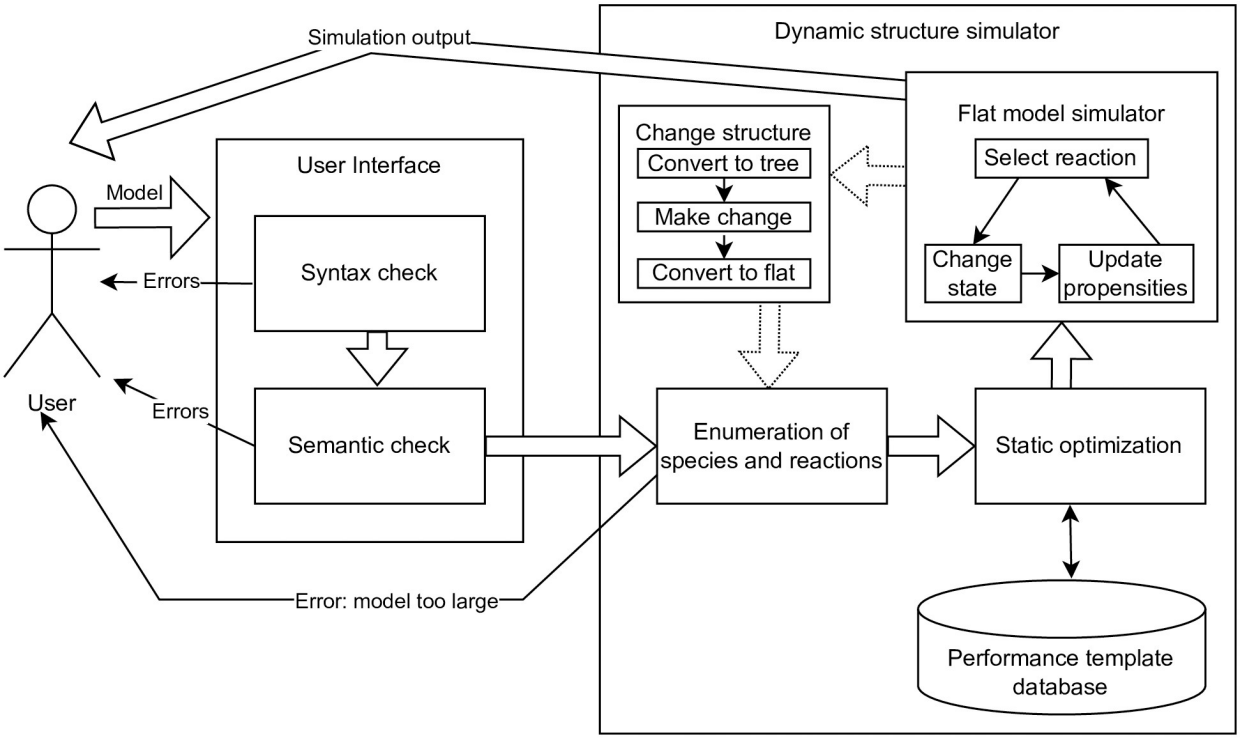


Fig 2. This Figure shows the flow of information between components when running a simulation. The user specifies the model, which is checked for syntactic and semantic consistency (like units or variable names). This is done through interaction with the web editor or command line interface. If these checks fail, errors are returned. Otherwise, the simulation loop starts by enumerating all possible species and reactions within the system into a flat representation. This can lead to errors if the system is too large. This flat representation is then optimized and put into the flat model (non-compartmental) simulator. Whenever a dynamic structural change is needed, the flattened model is transformed into a hierarchical, compartmental representation to execute those changes, and the loop starts again with the enumeration.

https://doi.org/10.1371/journal.pone.0312813.g002

We also incorporated a hybrid rule-based simulation method, allowing individual attributes to be simulated in a network-free manner (Section 2.1). Finally, through the use of recent advances in web technology (namely WebAssembly), we can provide a simple web editor that enables fast local execution using the same codebase (Section 2.5).

2.1 Language

ML-Rules 3 builds on and adapts the language ML-Rules for rule-based multi-level modeling of cell biological systems [6] and its formal semantics [47]. The model is comprised of

- constants that can be used as input parameters for simulation experiments,
- definitions of functions for simplification of repeated notations,
- the initial model state,
- rewriting rules, that can also change the structure of compartments and may use complex expressions for the rates and attribute values, and
- a definition of potential outputs of the model.

The rewriting rules have the form:

```
<left> -> <right> @ <rate>
```

We have a left side transformed (using the arrow) into the right side at a rate (following the @ symbol). The {} denote nesting relationships.

In the following, we describe some key aspects of ML-Rules and some slight changes compared to earlier implementations. The syntactic enhancements (with the exception of the network-free attributes) were not intended to improve the simulation performance but modeling in ML-Rules. A core aspect of the language design is the nature of the patterns on the rules' left side. Modeling languages based on graph rewriting, such as Kappa or BNGL, employ graph patterns. Modeling languages based on multiset rewriting, such as CSMMR or ML-Rules, employ term patterns whose matching can be considered a specific case of unification [50]. The matching relates to compartmental structures and attributes [6, 49]. Attributes can be addressed either by position (structural pattern matching) or by name.

For a species

$$\texttt{S(att\_1: int, att\_2: String)}$$

the previous ML-Rules versions required to list all attributes of a species:

$$\texttt{S(a1,a2) -> S(a1+1, a2)}$$

as attributes were identified by their position. Instead, we now have named attributes, where, in the case above, only the changed attribute needs to be listed:

$$\texttt{S() -> S(a1=S.a1+1)}$$

From a formal perspective, rule-based approaches implicitly match omitted attributes. For example, given a cell species that has two attributes denoting its phase and volume, the rule

$$\texttt{Cell(phase == 1) -> Cell(phase=2)}$$

would be implicitly extended to

$$\texttt{Cell(phase == 1, volume == v) -> Cell(phase=2, volume=v)}$$

to express that the volume does not change. Named attributes have been used in other tools like BioNetGen [38] or Chromar [51].

The use of named attributes is obviously particularly useful if species have many attributes. For example, a simulation model of the Baltic Cod, which was developed in a previous version of ML-Rules, relied on structural pattern matching [52], and is available at http://github.com/Baltic-Cod/EBC_IBM/blob/main/Basic_asph/Basic_asphyx.mlrj, has been rewritten using ML-rules 3 to exploit the more efficient execution. This also resulted in a more succinct representation due to the named attributes http://mlrules.pages.dev/gm/4/day.

Typically, the simulation engine for rule-based models transforms rules into reaction networks by enumerating all possible values of attributes in advance to speed up the actual simulation [53]. Attributes that may assume many different and potentially unbound possible values are a challenge. In the case of unbound continuous attribute values, e.g., if the size of a lipid raft changes depending on the number of membrane proteins being aggregated within the raft or due to merging [54], this *in-advance-enumeration* becomes impossible. Even finite, categorical values can lead to a combinatoric explosion in the number of reactions and thus increase the reaction network size beyond tractable limits. This results in a *model too large* error in Fig 2. We have introduced specific types for this kind of attribute called network-free-integer and network-free-continuous. If an attribute is defined as one of these types, its values will not be explicitly enumerated before simulation execution (to generate the reaction network). For simulation, the attribute of type network free is equipped with a vector that stores the currently

existing attribute values when a reaction fires. Once a rule fires, the species is instantiated with appropriate values. The approach is similar to the network-free simulation in BioNetGen [55] or CSMMR [5], particularly to the hybrid approach introduced in [56] that combines network-free and network-based calculations. However, instead of specifying entire species as network-free species, the modeler declares individual attributes to be network-free, and the simulator handles only those attributes or combinations as network-free. This distinction between being executed as network-based or network-free applies only to attributes of species that do not form a compartment. All compartments are handled as individual entities, as compartments are typically characterized by an attribute of continuous type, i.e., the volume, and they may contain an arbitrary number of diverse species (which again might be attributed), so one compartment's state is very likely different from the next and is (including its attributes' values) treated individually.

Other adaptations compared to earlier ML-Rule implementations are motivated by further increasing readability. In the previous ML-Rules versions, only equality constraints on the attribute values could be expressed on the left side of the rule. For example,

$$\texttt{Cell(vol, M)} \texttt{ -> } \ldots \texttt{ if vol} > 100 \texttt{ than k1 else } 0$$

can be expressed in ML-Rules 3 as

$$\texttt{Cell(vol} > 100, \texttt{phase} == \texttt{M)} \texttt{ -> } \ldots \texttt{ @ k1}$$

Further adaptations aim to provide further support to define correct models. Similarly, as in other tools [57], modelers are now asked to assign units of measurement to numerical variables and constants, like micrometer, 1/second, and liter. The simulator performs the proper conversion and automatic type checking [58] (Fig 2). This means a meter is correctly added to centimeters but not seconds. In addition, we introduced an enum type, allowing the modeler to constrain the admissible string values for a specific attribute to a specific set. Type inference, as well as constraint checks, occur ahead of simulation execution.

2.2 Efficient handling of dynamic structure reactions

As discussed in Sections 1.1 and 1.3, dynamic changes in the system's structure are core to the ML-Rules modeling language. Their efficient execution has been a challenge in the past. Most simulation tools assume static compartmental structures, and their simulators have been optimized accordingly [36, 38, 57]. Integrating compartmental dynamics into these simulators without significant loss of efficiency has been identified as a daunting challenge [59].

One option to solve the problem of dynamic structural changes is to write a fully dynamic simulator. This simulator traverses the tree structure of the model after each reaction execution to instantiate new reactions and re-calculate propensities on demand. It also checks whether the dependency graph requires any updates. This is how a previous implementation of ML-Rules was realized [49] and what is shown in the left panel of Fig 3. However, compared to methods optimized for static networks [53], this has a significant overhead.

For ML-Rules 3, we implemented a new hybrid simulator approach shown on the right panel of Fig 3. During an additional analysis step, every reaction is analyzed and marked if its execution would result in a structural change. In the models we encountered, most reactions do not induce a structural change. These reactions are called regular reactions. A similar distinction between regular and complex reactions has been made when simulating part of the system using deterministic numerical integration methods [60] or by tau-leaping [61].

The primary mode of our simulator is running an SSA (Section 2.3) that executes only regular (non-dynamic) reactions. The model representation has been flattened, enabling static

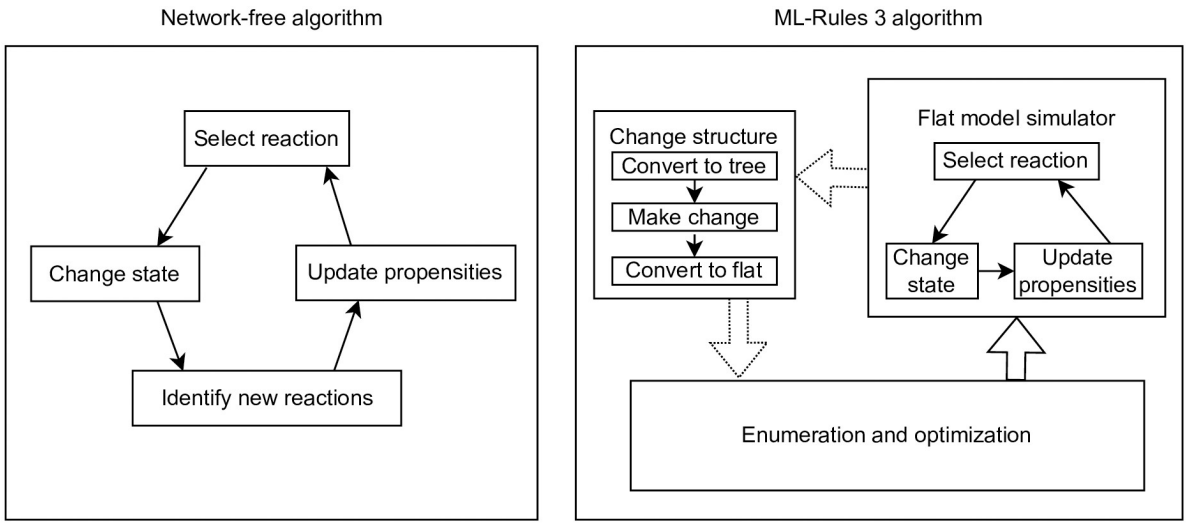


Fig 3. Comparison of the dynamic structure approaches. On the left, we have the previous network-free method. It is simpler to implement, but due to the dynamic structure, each step is significantly more costly in terms of performance. For example, neighborhood relations are dynamic, and therefore, the nesting tree structure needs to be traversed. Our approach (a subset of Fig 2) is on the right. We can utilize a faster flat (i.e., static) model simulator and have a costly but amortizing transformation and optimization process on the rare occasion of a structural change. For example, all relations can be encoded by static indices.

https://doi.org/10.1371/journal.pone.0312813.g003

index-based access to the model's state, i.e., every species' amount is stored in an array. Hierarchical relations are only preserved to the point where they are needed to reverse the system representation to a tree form. But they are neither accessed nor needed during the simulation as long as no transformation to a tree form is required. Therefore, we call this the *Flat model simulator* in Figs 2 and 3. When we encounter a structure-changing reaction, the model is transformed back into the tree representation reflecting the compartmental structures. The reaction is applied to this structural representation of the model. The resulting structured state is then transformed back into a flat representation, and processing is continued. This circular process is shown on the right side of Fig 3. This Figure shows the inner loop within the flat model simulator and the loop involving the dynamic structure changes. Every dynamic structure change requires an entire rebuild of the simulator, including re-enumerating all potential reactions and a do-over of the static optimization phase. Especially for larger systems, this can be relatively costly; however, as long as dynamics structural changes are rare, the costs amortize. We will analyze this in the evaluation section 3.

2.3 Stochastic simulation algorithm for static compartments

Based on our experience with different SSAs, we developed our variant, which is suited particularly well for typical problems in ML-Rules, where systems can be very large, e.g., for simulating multi-cellular models. Systems become particularly large when identical reactions are replicated across many different compartments. This is the *Flat model simulator* in Fig 2. The implementation is based on storing the propensities in a binary tree with cumulative sum tracking [62]. Each node in the tree corresponds to one potential reaction and stores two values: The propensity of that reaction and the sum of all propensities of its child nodes. This allows for updates of the total propensity sum with logarithmic complexity changing a single propensity. Furthermore, reaction selection (a weighted random choice based on the reaction propensities) can also be completed in logarithmic time complexity. Similar approaches are

used in other algorithms like the logarithmic direct method [63] or the simulator implementation for *Sπ@* [64].

An additional advantage of the tree-based approach is its improved numerical stability due to the fewer operations needed to update the propensities. Another way to minimize total propensity calculation time is by keeping track of the total propensity sum and only adding and subtracting the changes to reaction propensities after reaction execution as done by the Optimized Direct Method [32]. However, here, numerical errors from floating-point addition and subtraction accumulate over time and need to be dealt with. In our tree-based approach, numerical rounding errors only originate from the single summation, similar (arguably even better) to a single linear summation approach. The numerical error does not accumulate over time. It is independent of the number of steps and depends only on the number of reactions.

After a reaction has fired, multiple propensities in the tree must be updated. In principle, each update in the tree could be done individually based on the dependency graph via updating the cumulative sum until we reach the root. Instead, we roll these changes out in two phases. First, the node values are changed based on the dependency graph. Afterwards, the cumulative sums are updated as needed. If one were to update all sums individually, some nodes near the root might perform repeated summation updates. What total summation operations need to be done for each reaction is calculated ahead of time in the static optimization component (Fig 2). This computation has been accelerated via the use of bitsets. The generation of the dependency graph and the propensity tree must be efficient, as it needs to be redone after every non-regular reaction. Reaction selection is based on a weighted random choice based on the reaction propensities stored in the tree. The performance of reaction selection is improved by sorting the more likely reactions towards the root of the tree.

2.4 Performance templates

ML-Rules is an external domain-specific language. This means that all expressions are written in a custom text format. This format is parsed by the simulation tool. One of the main downsides of using an external domain-specific language is the significant computational overhead in the repeated evaluation of mathematical expressions. Generally, the more expressive and generic a domain-specific language is, the harder it is to execute efficiently. In a more constrained language, code and algorithmic variations that are tailored to the specific requirements of the application and hardware architecture are easier to achieve. However, a domain-specific language is typically more useful if it is more generic and expressive.

In [49], we developed specialized simulators for specific sub-classes of ML-Rules models, e.g., those that do not exploit compartmental dynamics. In [65], we developed an approach generating an entire simulator optimized for a specific model defined in a rule-based language, such as BioNetGen [37]. After parsing the model, custom C or Rust code was generated. This code was then compiled and optimized using existing compiler software, resulting in a high throughput performance. This technique (called *partial evaluation* or *Futamura projection*) [66, 67] is an established technique to deal with the problem that genericity does not go along well with achieving performance.

The central idea of partial evaluation is that a function with multiple inputs can be reduced to a simpler version if some of the inputs are known. For example, the power function $f(base, exponent)$ may be simplified to $f(base, 2) = base \cdot base$ for the case of a known exponent. The two core applications of this in the context of SSA are the dependency graph and the propensity calculations. For the dependency graph, data structures and loops can be omitted. With the propensity calculations, optimizations focus on reducing mathematical expressions and removing overhead from dynamic interpretation.

Adopting the approach developed in [65] for dynamic compartments appears impractical. Its performance gain relied to a large degree on optimizing the updates of the dependency network. With dynamic compartments, these reaction networks change during execution; thus, optimization (recompilation) would need to be repeated after each structural change. However, the (re-)compilation induces a significant overhead. Especially for large models (which is typical for ML-Rules models), the additional compilation steps on every model run increase runtime significantly, independently of model execution duration. Therefore, we developed an approach that combines dynamic interpretation and partial evaluation and does not require repeated compilation on every change. Our approach focuses on the rate evaluation (and thus propensity updates). This is costly for most dynamic compartmental models, as rate expressions involving dynamic compartments are typically complex. The complex expressions result in large abstract syntax trees (AST) that need to be parsed during execution. When using an interpreted or reflection-capable language like Java, code generation for these expressions can be introduced relatively simply [68]. For compiled languages such as Rust, a different approach is required. Our approach developed for ML-Rules 3 generates performance templates during the simulation and stores them for later reuse. Every time the simulator encounters a new AST, it checks whether generated code in the form of a performance template is available to replace this tree. Each performance template presents a partially evaluated AST and can be parameterized with numerical values (like model-specific constants) and indices (as used to identify involved species). If such an optimized previous version is found, it is used instead of the AST. If not, the simulator dynamically interprets the expression (by evaluating the AST). It generates an optimized code for future use and stores the corresponding performance template for later reuse. With every new model developed or recompilation of the model, the simulator can reuse previously created templates.

2.5 Web editor

The software tool is primarily designed as a command line tool. However, as a proof of concept, we also built a web-based version. The idea to run simulations on the web is not new and has been around almost as long as the web itself [69]. There have also been attempts at running stochastic simulations of biochemical models using a web interface [70].

However, the execution has been typically delegated to a server in the background or the cloud to enable an efficient simulation of models [71]. WebAssembly is a binary format recently developed that enables fast execution in web browsers [72]. WebAssembly is executed on a stack-based virtual machine similar to other native code. Additionally, it interoperates well with JavaScript. WebAssembly allows reasonable performance for simulation without installation in a web browser or major changes to the underlying simulator code [73].

As the Rust language can be compiled into WebAssembly, we can use the same code base for the simulator and only need to add a small frontend. We used the Rust Yew framework (http://yew.rs/) for this. A two-panel user interface is provided at http://mlrules.pages.dev, which includes a code editor with some syntax highlighting based on the Monaco editor from Visual Studio Code and another panel to display simulation results or any errors and instructions from the simulator.

A significant advantage of this WebAssembly approach is that there are next to no host costs. The simulation is executed on the end user's machine. The web page is static and has a size of roughly 10 Megabytes, which is very cheap or free to host compared to other approaches that run the simulation on the server. We also added the possibility to link to specific models. We use this in our case studies to provide links to the various models, which facilitates the reproduction of results and allows users to change and adapt models and conduct

basic experiments with no additional setup. As done throughout this paper, linking to an executable model with a custom URL is possible.

3 Evaluation

In this section, cell biological simulation models showcase the capabilities and performance of ML-Rules 3. All performance experiments were conducted using an intel i9-13900K CPU, running Rust 1.74. WebAssembly was run in Firefox version 120 using its spidermonkey engine.

3.1 Fission yeast model

The first case study is based on a multi-cellular model of fission yeast, including cell division and mating type switching depending on intracellular dynamics. The model is available at http://mlrules.pages.dev/yeast/300/min. The model (see Fig 4) has been adopted from the original ML-Rules publication [6] to show the expressiveness of ML-Rules. The fission yeast model includes an early cell cycle model [2].

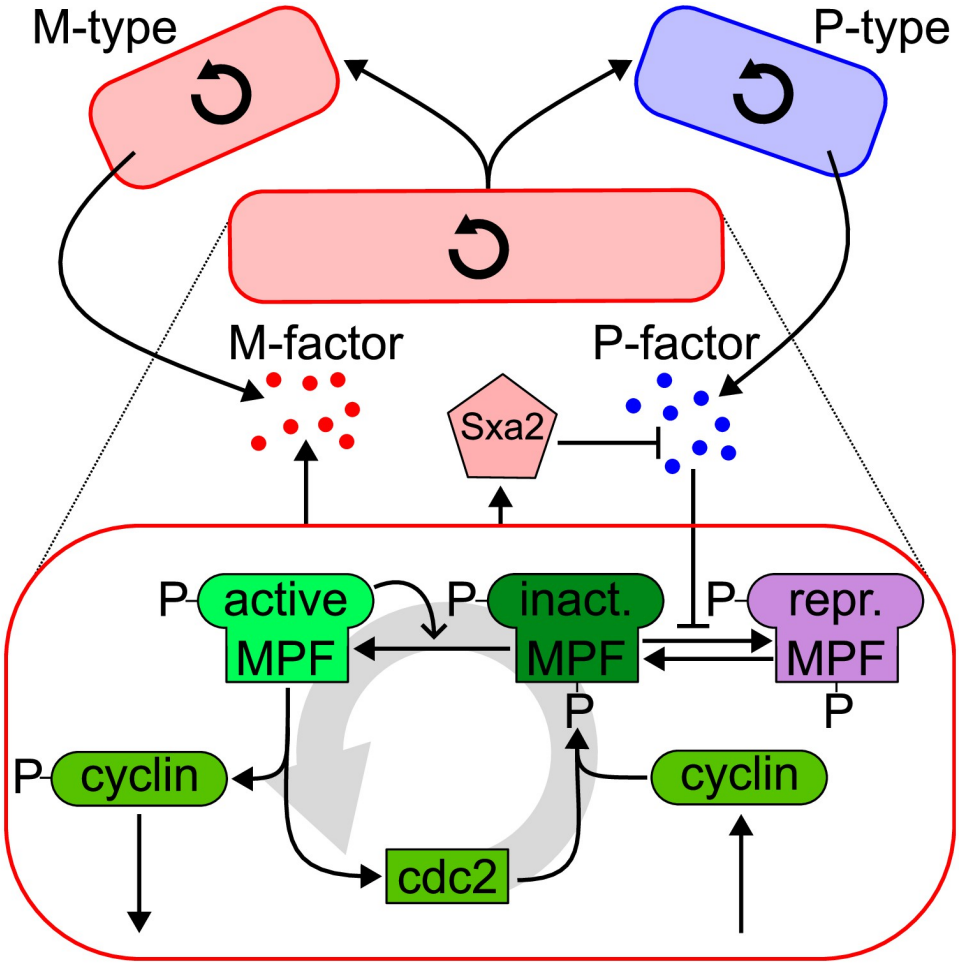
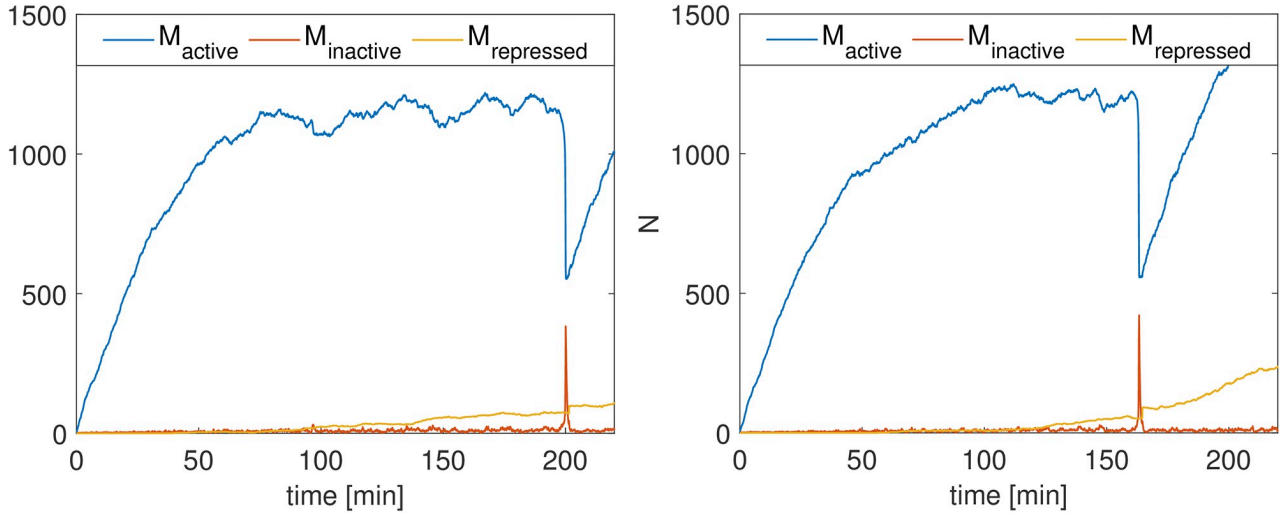


Fig 4. Fission yeast model. Inside the cell, proteins oscillate. Outside the cell, Sxa2 and pheromones (P and M-type) can inhibit the cell cycle of cells of the opposite mating type. Triggered by the cell cycle, a yeast cell can fission into two daughter cells.

https://doi.org/10.1371/journal.pone.0312813.g004



5. Two sample replications of the fission yeast model show the oscillation of active, inactive, and restricted MPF. At 200 minutes and 163 minutes, ›ectively, a fission event is triggered by the spike of inactive MPF.

ıs://doi.org/10.1371/journal.pone.0312813.g005

The cell cycle model consists of two proteins (cyclin and cdc2) that can form the maturation-promoting factor (MPF). MPF exists in three versions (active, inactive, and repressed), which oscillate with a cycle duration of approximately 200 minutes (see Fig 5). This oscillation triggers the cell to change between its phases (G1, SG2, and M), and eventually, a spike in inactive MPF causes cells in its M-phase to divide into two daughter cells. In addition, cells are characterized by a mating type (P or M) that might differ in one daughter cell from the type of the mother cell. Based on the mating type, cells produce and secrete pheromones (M- or P-factor) to inhibit the cell cycle of the opposite mating type cells by changing MPF from its inactive to the repressed variant. M-type cells also release P-factor-specific protease (Sxa2) that inhibits the P-factor pheromone.

This model relies on a unique combination of ML-Rules features and its expressiveness. Individual cells are defined as compartments that frequently divide. Compartments and proteins are equipped with attributes, e.g., to denote the cell cycle phase, the cyclin's phosphorylation state, or the cell's volume. The cells secrete pheromones into the extracellular environment. They influence the cell cycle of other cells of the opposite mating type. The kinetics depend on rate factors that require complex expressions, e.g., a Hill-type sigmoidal response curve defines the MPF repression (depending on the number of pheromones).

We have used this model as a performance benchmark. The reaction throughput rates for a run until 1000 minutes (simulation time) are shown in Table 1. ML-Rules 3 is significantly

Table 1. Run time and reactions per second for different simulators. The fission yeast model is executed until 1000 minutes (20 replications). The values in parenthesis refer to executing ML-Rules 3 as WebAssembly code. The ML-Rules 2 implementation of the model contains ten species and 20 rules. The ML-Rules 3 version of the model consists of 7 species and 20 rules. The model starts with two cell compartments and ends with about 12. During the simulation time, the model undergoes about 10 structural changes and half a million static reactions.

simulator	runtime [s]	throughput [1000/s]
ML-Rules 2	12.4	40.8
ML-Rules 3	0.124 (0.190)	4190 (2690)
ML-Rules 3—network free	0.321 (0.352)	1590 (1440)
ML-Rules 3—w/o templates	0.259 (0.331)	1950 (1530)

https://doi.org/10.1371/journal.pone.0312813.t001

faster than the previous Java implementation, ML-Rules 2. If performance templates are used, we observe a roughly 2-order speedup. Even without the templates, the performance is 26 to 18 times faster. The 18x speedup is observed when enabling network-free attributes (see Section 2.1). The model does not use network-free attributes; even only enabling this capability introduces more branching in the critical code and slows down execution. We also tested the WebAssembly version. The data for this plot can be generated locally by visiting http://mlrules.pages.dev/benchmark/yeast/3/1000. The first number is the number of replications, and the second is the longest test duration in minutes of simulation time. The WebAssembly has a performance penalty of only about 10% to 50%.

3.2 mRNA delivery model

Our second case study is centered around the delivery of mRNA into cells. Understanding how to deliver mRNA into cells is crucial for its use as a drug or vaccine [21]. Ligon et al. [74] published a simulation model capable of simulating the mRNA delivery based on lipoplexes (small lipid spheres containing mRNA).

In their model, shown in Fig 6, lipoplexes are present within the extra-cellular environment for an hour before being removed by an event mimicking the cell's washing. During this time, clathrin-coated pits containing one or more lipoplexes form at the cell's surface. Via the invagination of the plasma membrane, endosomes are formed, and the lipoplexes enter the cell. Inside the cell, the endosome releases the lipoplexes, which then unpack their mRNA. Afterward, the mRNA can be translated into proteins.

With their model, the authors could gain insight into mRNA delivery by lipoplexes and the dose-response relationship. However, they also state a few simplifications and workarounds needed in their model, as the used modeling and simulation method only supported static compartments. One simplification is that all lipoplexes carry the same amount of mRNA in the model. In wet lab experiments, this number has been found to vary due to the different sizes of the lipoplexes [74, 75].

One possibility to model lipoplexes more realistically is to model each lipoplex as a compartment that contains the mRNA and enters the cell (which is also represented as a compartment). For this, support of dynamic compartments is required. Another possibility is to model

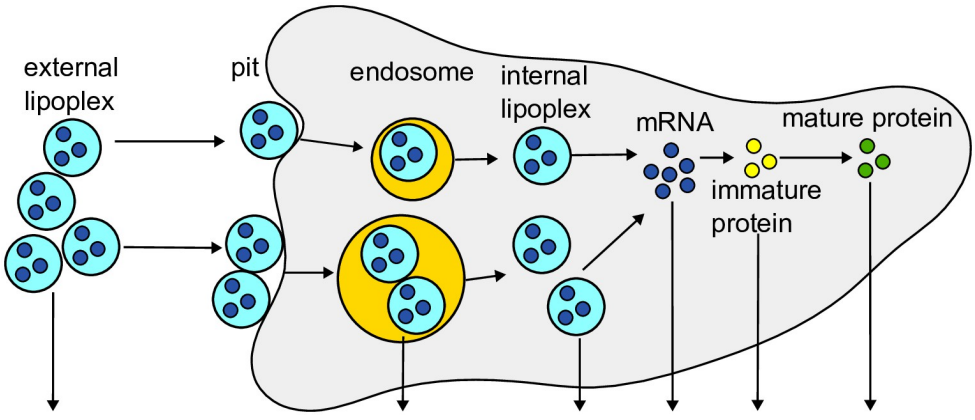
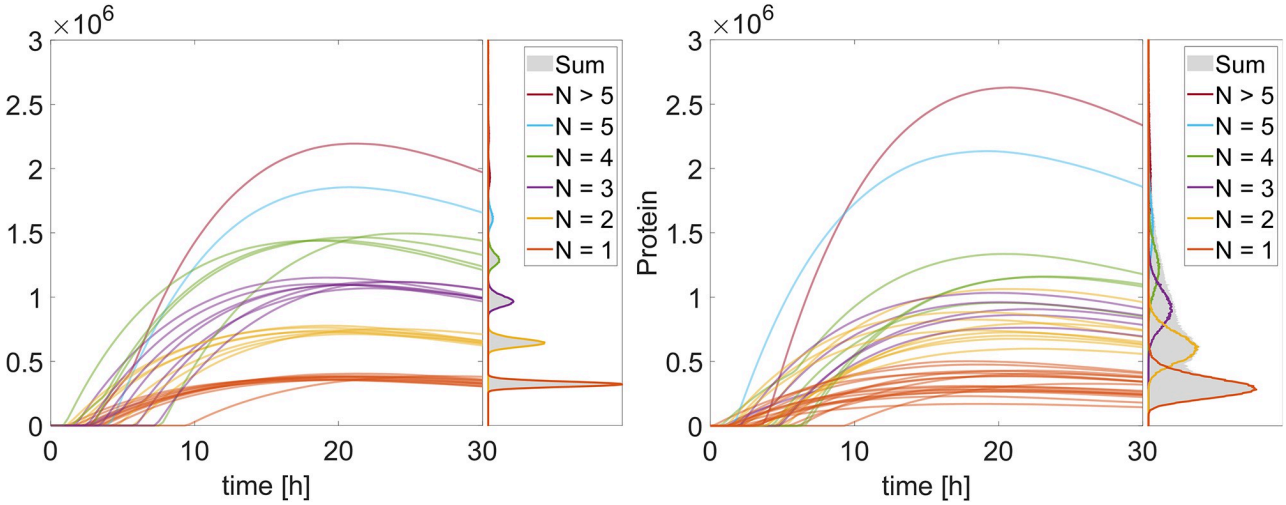


Fig 6. Sketch of the mRNA delivery model. Lipoplexes are present in the cell's environment and can accumulate in pits in the cell membrane. The lipoplexes in the pit can enter the cell via endocytosis, and the endosome can lyse to release the lipoplexes into the cell. Once they unpack their mRNA, the mRNA gets translated to proteins. The arrows that point down indicate the degradation of the species.

https://doi.org/10.1371/journal.pone.0312813.g006



7. Protein amounts over time for the original (left) and modified (right) mRNA delivery model. The time courses are for 100 replications. The ors indicate the number of lipoplexes that unpack their mRNA in the cell. The histograms next to the time courses show the distribution of proteins after ıours without the cells that express no protein (1000000 replications). Runs where no lipoplexes unpack their mRNA in the cell and no proteins are erated are not shown.

s://doi.org/10.1371/journal.pone.0312813.g007

the amount of mRNA a lipoplex contains as a specific attribute of type integer. Both solutions are possible in ML-Rules but were not possible in the tool(s) used by the authors. This led to the simplification of assuming a fixed number of mRNA per lipoplex (i.e., 350) in the model. As a result, the simulation of the protein expression in the original model shows narrow bands, depending on the number of lipoplexes that could enter the cell and unpack their mRNA (see Fig 7 left), which is not the case in the wet-lab data [75]. The authors tried to use the modeling and simulation tool SPim [76], which is based on the stochastic π calculus [77] and allows to assign an attribute to a lipoplex that states how much mRNA it carries and unpacks this amount into the cell. However, they ran into performance issues, which made it impossible to use SPim for their study (see supplement TextS001 p. 7 from [74]).

As stated above in ML-Rules, this simplification is unnecessary, and a varying number of mRNA can be assigned to the lipoplexes modeled as compartments. The modified model can be found under http://mlrules.pages.dev/lipoplex_ext/30/h. At the beginning of the simulation, the simulation state is set to contain one cell and 200 lipoplexes containing mRNA. The amount of mRNA per lipoplex (L) is calculated based on the lipoplex size sampled from a normal distribution. The lipoplex can then move into the cell, like in the original model. Inside the cell, the lipoplex compartment L can unpack its content (?solL).

```
Cell{L{?solL}+?solC} -> Cell{?solL+?solC} @ kU;
```

The ?solL denotes the content of the lipoplex, here a population of mRNA, and ?solC is the cell's content, including other lipoplexes, mRNA, and proteins. Alternatively, we could have modeled the mRNA as an attribute of type integer. In such an implementation, the lipoplex would not be a compartment carrying the mRNA but a simple species equipped with an attribute (NmRNA) that denotes the amount of mRNA inside. The unpack rule shown above would change:

```
Cell{L+?solC} -> Cell{(L.NmRNA) mRNA+?solC} @ kU;
```

A second minor simplification is that lipoplexes can unpack their mRNA not only in the cell but also in the endosome, where the unpacked mRNA can start to deteriorate. This mechanism, called the "fully nested transfection model" by the authors (see Fig 8 in Ligon et al. [74]), is missing in the original model.

In the ML-Rules 3 model, it is realized by the two reactions:

```
Cell{E{L{?solL}+?solE}+?solC} -> Cell{E{?solL+?solE}+?solC} @ kU;
```

for the unpacking inside the endosome, and

```
E{mRNA : m+?solE} -> E{?solE} @ dM*m;
```

for the degradation of mRNA in the endosome.

Besides the simplification in the original model, some reactions, like the formation of pits, are lengthy to write down. Due to the lack of attributes or dynamic compartments, the number of lipoplexes that reside in a pit is stated in the name of the species. The original model uses ten pit species (P1—P10) denoting 1 to 10 lipoplexes inside the pit. Consequently, ten reactions need to be specified for all P_i regarding the formation of pits, the endocytosis, the lysis, and the degradation of pits. This workaround only works for a low number of P-species. The growing number of reactions makes writing the model down for larger numbers increasingly harder. In ML-Rules 3, the dynamic nesting allows us to write down reactions more compactly. For example, the ten endocytosis reactions in the original model:

```
P1 -> E1
...
P10 -> E10
```

translate into a single rule in ML Rules 3.

```
Cell(?solC)+P{?solP -> Cell{E{?solP}+?solC}
```

By applying the abovementioned changes, the model produces more realistic results, as can be seen by the protein expression over time in Fig 7. As stated above, the narrow bands as observed in the original model (Fig 7 left) are not observed in the wet lab experiments [74, 75]. By varying the mRNA number inside the lipoplexes, these bands in the protein expression broaden and overlap (Fig 7 right), resulting in a more realistic model behavior.

Finally, we have a look at the model's runtime performance. Therefore, we compared the runtime for executing the original model written in COPASI (version 4.41) and its ML-Rules equivalent (available at http://mlrules.pages.dev/lipoplex_orig. The original COPASI model consists of 48 reactions and 26 species compared to the ML-Rules model with its 12 rules and 8 species. This is due to the manual unrolling of the underlying nesting processes and illustrates ML-Rules' expressiveness, which also results in succinct models. We measured the runtime for 1000 replications.

As the simulation model contains events to describe the removal of external lipoplexes, only the Direct Method implementation within COPASI is able to execute the entire simulation model. This is not a limitation of the method but of the implementations. Timed events can be added relatively easily to most methods as an implementation feature. The difference in methods should be considered when interpreting the runtime measurements. For larger systems, the direct methods used generally perform worse than a more optimized logarithmic method, like the next reaction method.

We found the average runtime for a single execution to be 4.8 seconds (see Table 2). However, in two-thirds of the simulation runs, no lipoplex can unpack their mRNA into the cell,

Table 2. Run time for different simulators. The mRNA delivery model is executed until 30 hours (1000 replications). The ML-Rules 3 model is initialized with 201 compartments (1 cell and 200 lipoplexes) and ends with one compartment. On average, about 207 structural changes are executed during a simulation run.

simulator	runtime [s]		
	20% quantile	average	80% quantile
Copasi	0.13	4.8	11.2
ML-Rules 3	0.59	0.6	0.84

https://doi.org/10.1371/journal.pone.0312813.t002

and consequently, no proteins are created. The 20% quantile takes 0.13 seconds, and the 80% quantile is 11.2 seconds. The equivalent simplified model in ML-Rules takes only about 0.6 seconds on average but 0.59 seconds for 20% and 0.84 seconds for the 80% quantile, respectively. Most of the runtime (87% on average) in the ML-Rules model is spent on a dynamic structure reactions, something that COPASI does not need to consider. When no mRNA is unpacked into the cell, and no proteins are created, a simulation run has as many static as dynamic structure reactions (about 200). When one lipoplex unpacks its mRNA into the cell, about 200 dynamic structure reactions happen, and about 2 million static structure reactions occur. Nevertheless, the dynamic structure reactions need about five times longer to be executed. We find similar runtimes for the adapted version of the model in ML-Rules.

4 Conclusion

We built ML-Rules version 3, making concise formulations of dynamic structure models run with high performance. A performance-oriented implementation of various data structures, including a partial summation tree, made this possible. We also made some changes to the language, like introducing named attributes and units of measurement. We found that this implementation outperforms the previous version of ML-Rules by two orders of magnitude. The evaluation of the simulator is based on two biological case studies. First, we used a fission yeast model that was also used in the original ML-Rules publication to show that we have a similar expressiveness but a higher performance than the previous implementation. Second, we rebuilt and extended an mRNA delivery model and showed how using dynamic compartments needs fewer simplifications than the original model that uses a static compartmental approach. The extended model matches the wet-lab data more closely, allowing a more compact notation and better performance. Finally, we built a prototypical web-based simulator using the same source code compiled to WebAssembly that can run locally on the end user machine. WebAssembly's ease of deployment and development using existing code bases and competitive performance are promising. We see further opportunities for simulation tool developers to make their software more accessible using this technology.

Our investigation has also raised some questions for future research. Currently, the network-free execution part of the model is determined by attributes explicitly defined as network-free. However, possibly a larger portion of the model could benefit from a network-free execution. To learn during simulation which part of the model to execute most efficiently in a network-based or network-free manner, possibly reinforcement learning approaches could be adapted [78].

The main simulator for ML-Rules 3 now conforms to standard SSA, with the potential exception of more complex rate expressions. It is easier to integrate with previous research on more advanced SSA variants like the partial propensity method [79] or approximate methods like advanced tau leaping [80]. Preliminary experiments with approximate tau leaping showed significant improvement for some models. However, frequently, the limiting factor is the

repeated structural conversion. Moving forward, only a subset of the simulator could be transformed if changes to the dynamic structure are localized. It would also be possible to integrate the dynamic structure more closely with the simulator, at the cost of some performance for the regular transitions. The simulator is currently applied in three different simulation studies, i.e., studying the fission and fusion of mitochondria, bone remodeling processes, and the role of endocytosis in cellular signaling.

Acknowledgments

Fig 1 was created with BioRender.com.

Author Contributions

Conceptualization: Till Köster, Tom Warnke, Adelinde Uhrmacher.

Data curation: Till Köster, Philipp Henning.

Formal analysis: Till Köster, Philipp Henning.

Funding acquisition: Adelinde Uhrmacher.

Investigation: Till Köster, Philipp Henning.

Methodology: Till Köster, Philipp Henning.

Project administration: Adelinde Uhrmacher.

Resources: Adelinde Uhrmacher.

Software: Till Köster.

Supervision: Adelinde Uhrmacher.

Validation: Till Köster, Philipp Henning.

Visualization: Till Köster, Philipp Henning.

Writing – original draft: Till Köster, Philipp Henning, Tom Warnke, Adelinde Uhrmacher.

Writing – review & editing: Till Köster, Philipp Henning, Adelinde Uhrmacher.

References

1. Weng G, Bhalla US, Iyengar R. Complexity in biological signaling systems. Science. 1999; 284 (5411):92–96. https://doi.org/10.1126/science.284.5411.92 PMID: 10102825
2. Tyson JJ. Modeling the cell division cycle: cdc2 and cyclin interactions. Proceedings of the National Academy of Sciences. 1991; 88(16):7328–7332. https://doi.org/10.1073/pnas.88.16.7328 PMID: 1831270
3. Regev A, Panina EM, Silverman W, Cardelli L, Shapiro E. BioAmbients: an abstraction for biological compartments. Theoretical Computer Science. 2004; 325(1):141–167. https://doi.org/10.1016/j.tcs.2004.03.061
4. John M, Lhoussaine C, Niehren J, Versari C. Biochemical Reaction Rules with Constraints. In: Proceedings of the 20th European Symposium on Programming, ESOP 2011. Berlin, Heidelberg: Springer-Verlag; 2011. p. 338–357.
5. Oury N, Plotkin GD. Multi-level modelling via stochastic multi-level multiset rewriting. Mathematical Structures in Computer Science. 2013; 23(2):471–503. https://doi.org/10.1017/s0960129512000199
6. Maus C, Rybacki S, Uhrmacher AM. Rule-based multi-level modeling of cell biological systems. BMC Systems Biology. 2011; 5(1):166. https://doi.org/10.1186/1752-0509-5-166 PMID: 22005019
7. Faeder JR. Toward a comprehensive language for biological systems. BMC Biology. 2011; 9(1). https://doi.org/10.1186/1741-7007-9-68 PMID: 22005092

8. Tokarev AA, Alfonso A, Segev N. In: Overview of Intracellular Compartments and Trafficking Pathways. Springer New York; 2009. p. 3–14. Available from: http://dx.doi.org/10.1007/978-0-387-93877-6_1.
9. Harris LA, Hogg JS, Faeder JR. Compartmental rule-based modeling of biochemical systems. In: Proceedings of the 2009 Winter Simulation Conference. WSC '09. Austin, Texas: IEEE; 2009. p. 908–919.
10. Liu J, Xiao Q, Xiao J, Niu C, Li Y, Zhang X, et al. Wnt/β-catenin signalling: function, biological mechanisms, and therapeutic opportunities. Signal Transduction and Targeted Therapy. 2022; 7(1). https://doi.org/10.1038/s41392-021-00762-6 PMID: 34980884
11. Budde K, Smith J, Wilsdorf P, Haack F, Uhrmacher AM. Relating simulation studies by provenance–Developing a family of Wnt signaling models. PLoS Computational Biology. 2021; 17(8):1–21. https://doi.org/10.1371/journal.pcbi.1009227 PMID: 34351901
12. MacLean AL, Rosen Z, Byrne HM, Harrington HA. Parameter-free methods distinguish Wnt pathway models and guide design of experiments. Proceedings of the National Academy of Sciences. 2015; 112 (9):2652–2657. https://doi.org/10.1073/pnas.1416655112 PMID: 25730853
13. Haack F, Lemcke H, Ewald R, Rharass T, Uhrmacher AM. Spatio-temporal Model of Endogenous ROS and Raft-Dependent WNT/Beta-Catenin Signaling Driving Cell Fate Commitment in Human Neural Progenitor Cells. PLOS Computational Biology. 2015; 11(3):1–28. https://doi.org/10.1371/journal.pcbi.1004106 PMID: 25793621
14. Hucka M, Bergmann FT, Chaouiya C, Dräger A, Hoops S, Keating SM, et al. The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 2 Core Release 2. Journal of Integrative Bioinformatics. 2019; 16(2). https://doi.org/10.1515/jib-2019-0021 PMID: 31219795
15. Duso L, Zechner C. Stochastic reaction networks in dynamic compartment populations. 2020; 117 (37):22674–22683. https://doi.org/10.1073/pnas.2003734117
16. Uhrmacher AM. Dynamic structures in modeling and simulation: a reflective approach. ACM Trans Model Comput Simul. 2001; 11(2):206–232. https://doi.org/10.1145/384169.384173
17. Kolch W, Halasz M, Granovskaya M, Kholodenko BN. The dynamic control of signal transduction networks in cancer cells. Nature Reviews Cancer. 2015; 15(9):515–527. https://doi.org/10.1038/nrc3983 PMID: 26289315
18. Basson MA. Signaling in Cell Differentiation and Morphogenesis. Cold Spring Harbor Perspectives in Biology. 2012; 4(6):a008151–a008151. https://doi.org/10.1101/cshperspect.a008151 PMID: 22570373
19. Sorkin A, von Zastrow M. Endocytosis and signalling: intertwining molecular networks. Nature Reviews Molecular Cell Biology. 2009; 10(9):609–622. https://doi.org/10.1038/nrm2748 PMID: 19696798
20. Huotari J, Helenius A. Endosome maturation. The EMBO Journal. 2011; 30(17):3481–3500. https://doi.org/10.1038/emboj.2011.286 PMID: 21878991
21. Hou X, Zaks T, Langer R, Dong Y. Lipid nanoparticles for mRNA delivery. Nature Reviews Materials. 2021; 6(12):1078–1094. https://doi.org/10.1038/s41578-021-00358-0 PMID: 34394960
22. Parchekani J, Allahverdi A, Taghdir M, Naderi-Manesh H. Design and simulation of the liposomal model by using a coarse-grained molecular dynamics approach towards drug delivery goals. Scientific Reports. 2022; 12(1). https://doi.org/10.1038/s41598-022-06380-8 PMID: 35149771
23. Damgaard TC, Højsgaard E, Krivine J. Formal cellular machinery. Electronic Notes in Theoretical Computer Science. 2012; 284:55–74. https://doi.org/10.1016/j.entcs.2012.05.015
24. Cardelli L. Brane calculi: Interactions of biological membranes. In: International Conference on Computational Methods in Systems Biology. Springer; 2004. p. 257–278.
25. Priami C, Quaglia P. Beta binders for biological interactions. In: Computational Methods in Systems Biology: International Conference CMSB 2004, Paris, France, May 26-28, 2004, Revised Selected Papers. Springer; 2005. p. 20–33.
26. Westermann B. Mitochondrial fusion and fission in cell life and death. Nature Reviews Molecular Cell Biology. 2010; 11(12):872–884. https://doi.org/10.1038/nrm3013 PMID: 21102612
27. Bozzuto G, Molinari A. Liposomes as nanomedical devices. International Journal of Nanomedicine. 2015; 10:975–999. https://doi.org/10.2147/IJN.S68861 PMID: 25678787
28. V'kovski P, Kratzel A, Steiner S, Stalder H, Thiel V. Coronavirus biology and replication: implications for SARS-CoV-2. Nature Reviews Microbiology. 2021; 19(3):155–170. https://doi.org/10.1038/s41579-020-00468-6 PMID: 33116300
29. Gillespie DT. Stochastic Simulation of Chemical Kinetics. Annual Review of Physical Chemistry. 2007; 58(1):35–55. https://doi.org/10.1146/annurev.physchem.58.032806.104637 PMID: 17037977
30. Anderson WJ. Continuous-Time Markov Chains: An Applications-Oriented Approach (Springer Series in Statistics). Springer; 1991.
31. Gillespie DT. Exact stochastic simulation of coupled chemical reactions. The Journal of Physical Chemistry. 1977; 81(25):2340–2361. https://doi.org/10.1021/j100540a008

32. Cao Y, Li H, Petzold L. Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. The Journal of Chemical Physics. 2004; 121(9):4059–4067. https://doi.org/10.1063/1.1778376 PMID: 15332951

33. Gibson MA, Bruck J. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. The Journal of Physical Chemistry A. 2000; 104(9):1876–1889. https://doi.org/10.1021/jp993732q

34. Faeder JR, Blinov ML, Hlavacek WS. Rule-based modeling of biochemical systems with BioNetGen. Systems biology. 2009; p. 113–167. https://doi.org/10.1007/978-1-59745-525-1_5 PMID: 19399430

35. Danos V, Laneve C. Formal molecular biology. Theoretical Computer Science. 2004; 325(1):69–110. https://doi.org/10.1016/j.tcs.2004.03.065

36. Boutillier P, Maasha M, Li X, Medina-Abarca HF, Krivine J, Feret J, et al. The Kappa platform for rule-based modeling. Bioinformatics. 2018; 34(13):i583–i592. https://doi.org/10.1093/bioinformatics/bty272 PMID: 29950016

37. Blinov ML, Faeder JR, Goldstein B, Hlavacek WS. BioNetGen: Software for Rule-based Modeling of Signal Transduction Based on the Interactions of Molecular Domains. Bioinformatics. 2004; 20(17):3289–3291. https://doi.org/10.1093/bioinformatics/bth378 PMID: 15217809

38. Harris LA, Hogg JS, Tapia JJ, Sekar JAP, Gupta S, Korsunsky I, et al. BioNetGen 2.2: advances in rule-based modeling. Bioinformatics. 2016; 32(21):3366–3368. https://doi.org/10.1093/bioinformatics/btw469 PMID: 27402907

39. Pedersen M, Phillips A, Plotkin GD. A High-Level Language for Rule-Based Modelling. PLOS ONE. 2015; 10(6):1–26. https://doi.org/10.1371/journal.pone.0114296 PMID: 26043208

40. John M, Lhoussaine C, Niehren J, Uhrmacher AM. The attributed pi calculus. In: International Conference on Computational Methods in Systems Biology. Springer; 2008. p. 83–102.

41. Danos V, Feret J, Fontana W, Krivine J. Scalable Simulation of Cellular Signaling Networks. In: Programming Languages and Systems. Springer Berlin Heidelberg; 2007. p. 139–157.

42. Bistarelli S, Cervesato I, Lenzini G, Marangoni R, Martinelli F. On Representing Biological Systems through Multiset Rewriting. In: Computer Aided Systems Theory—EUROCAST 2003. Springer Berlin Heidelberg; 2003. p. 415–426.

43. Cavaliere M, Sedwards S. Modeling and Simulating Biological Processes with Stochastic Multiset Rewriting. In: Nicol DM, Priami C, Nielson HR, Uhrmacher AM, editors. Simulation and Verification of Dynamic Systems. No. 06161 in Dagstuhl Seminar Proceedings. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany; 2006.Available from: http://drops.dagstuhl.de/opus/volltexte/2006/706.

44. Eker S. Associative-Commutative Rewriting on Large Terms. In: Rewriting Techniques and Applications. Springer Berlin Heidelberg; 2003. p. 14–29.

45. Dundua B, Kutsia T, Marin M. Variadic equational matching in associative and commutative theories. Journal of Symbolic Computation. 2021; 106:78–109. https://doi.org/10.1016/j.jsc.2021.01.001

46. Marin M, Tepeneu D. Programming with sequence variables: The Sequentica package. In: Challenging The Boundaries Of Symbolic Computation: (With CD-ROM). World Scientific; 2003. p. 17–24.

47. Warnke T, Helms T, Uhrmacher AM. Syntax and Semantics of a Multi-Level Modeling Language. SIGSIM PADS'15. New York, NY, USA: Association for Computing Machinery; 2015. p. 133–144. Available from: https://doi.org/10.1145/2769458.2769467.

48. Oury N, Plotkin GD. Coloured stochastic multilevel multiset rewriting. In: Proceedings of the 9th International Conference on Computational Methods in Systems Biology—CMSB 2011. ACM Press; 2011.

49. Helms T, Warnke T, Maus C, Uhrmacher AM. Semantics and Efficient Simulation Algorithms of an Expressive Multi-Level Modeling Language. ACM Transactions on Modeling and Computer Simulation. 2017; 27(2):8:1–8:25. https://doi.org/10.1145/2998499

50. Warnke T, Uhrmacher AM. Nonlinear pattern matching in rule-based modeling languages. In: Computational Methods in Systems Biology: 19th International Conference, CMSB 2021, Bordeaux, France, September 22–24, 2021, Proceedings 19. Springer; 2021. p. 198–214.

51. Honorato-Zimmer R, Millar AJ, Plotkin GD, Zardilis A. Chromar, a language of parameterised agents. Theoretical Computer Science. 2019; 765:97–119. https://doi.org/10.1016/j.tcs.2017.07.034

52. Pierce ME, Warnke T, Krumme U, Helms T, Hammer C, Uhrmacher AM. Developing and validating a multi-level ecological model of eastern Baltic cod (Gadus morhua) in the Bornholm Basin—a case for domain-specific languages. Ecological Modeling. 2017; 361:49–65. https://doi.org/10.1016/j.ecolmodel.2017.07.012

53. Gupta A, Mendes P. An Overview of Network-Based and -Free Approaches for Stochastic Simulation of Biochemical Systems. Computation. 2018; 6(1). https://doi.org/10.3390/computation6010009 PMID: 29938118

54. Simons K, Sampaio JL. Membrane Organization and Lipid Rafts. Cold Spring Harbor Perspectives in Biology. 2011; 3(10):a004697–a004697. https://doi.org/10.1101/cshperspect.a004697 PMID: 21628426

55. Sneddon MW, Faeder JR, Emonet T. Efficient modeling, simulation and coarse-graining of biological complexity with NFsim. Nature Methods. 2010; 8(2):177–183. https://doi.org/10.1038/nmeth.1546 PMID: 21186362

56. Hogg JS, Harris LA, Stover LJ, Nair NS, Faeder JR. Exact hybrid particle/population simulation of rule-based models of biochemical systems. PLoS computational biology. 2014; 10(4):e1003544. https://doi.org/10.1371/journal.pcbi.1003544 PMID: 24699269

57. Hoops S, Sahle S, Gauges R, Lee C, Pahle J, Simus N, et al. COPASI—a complex pathway simulator. Bioinformatics. 2006; 22(24):3067–3074. https://doi.org/10.1093/bioinformatics/btl485 PMID: 17032683

58. Kojima K, Kinoshita M, Suenaga K. Generalized homogeneous polynomials for efficient template-based nonlinear invariant synthesis. vol. 747. Elsevier BV; 2018. p. 33–47. Available from: http://dx.doi.org/10.1016/j.tcs.2018.06.005.

59. Thompson-Walsh CD, Hayman J, Winskel G. Containment in Rule-Based Models. Electronic Notes in Theoretical Computer Science. 2012; 284:125–137. https://doi.org/10.1016/j.entcs.2012.05.019

60. Helms T, Wilsdorf P, Uhrmacher AM. Hybrid simulation of dynamic reaction networks in multi-level models. In: Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation; 2018. p. 133–144.

61. Helms T, Luboschik M, Schumann H, Uhrmacher AM. An Approximate Execution of Rule-Based Multi-level Models. In: Computational Methods in Systems Biology. Springer Berlin Heidelberg; 2013. p. 19–32. Available from: http://dx.doi.org/10.1007/978-3-642-40708-6_3.

62. Köster T, Uhrmacher AM. Handling Dynamic Sets of Reactions in Stochastic Simulation Algorithms. In: Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. SIGSIM-PADS'18. ACM; 2018. p. 161–164. Available from: http://dx.doi.org/10.1145/3200921.3200943.

63. Li H, Petzold LR. Logarithmic Direct Method for Discrete Stochastic Simulation of Chemically Reacting Systems; 2006. Available from: https://api.semanticscholar.org/CorpusID:7936447.

64. Versari C, Busi N. Efficient Stochastic Simulation of Biological Systems with Multiple Variable Volumes. Electronic Notes in Theoretical Computer Science. 2008; 194(3):165–180. https://doi.org/10.1016/j.entcs.2007.12.012

65. Köster T, Warnke T, Uhrmacher AM. Generating Fast Specialized Simulators for Stochastic Reaction Networks via Partial Evaluation. ACM Transactions on Modeling and Computer Simulation (TOMACS). 2022; 32(2):1–25. https://doi.org/10.1145/3485465

66. Futamura Y. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. Higher Order Symbol Comput. 1999; 12(4):381–391. https://doi.org/10.1023/A:1010095604496

67. Leißa R, Boesche K, Hack S, Pérard-Gayot A, Membarth R, Slusallek P, et al. AnyDSL: A partial evaluation framework for programming high-performance libraries. 2018; 2(OOPSLA). https://doi.org/10.1145/3276489

68. Meyer T, Helms T, Warnke T, Uhrmacher AM. Runtime Code Generation for Interpreted Domain-Specific Modeling Languages. In: Winter Simulation Conference (WSC 2018). IEEE; 2018. p. 605–616. Available from: https://ieeexplore.ieee.org/document/8632545.

69. Fishwick PA. Web-Based Simulation: Some Personal Observations. WSC'96. USA: IEEE Computer Society; 1996. p. 772–779. Available from: https://doi.org/10.1145/256562.256807.

70. Ivanov S, Rogojin V, Azimi S, Petre I. Webrsim: A web-based reaction systems simulator. Enjoying Natural Computing: Essays Dedicated to Mario de Jesús Pérez-Jiménez on the Occasion of His 70th Birthday. 2018; p. 170–181. https://doi.org/10.1007/978-3-030-00265-7_14

71. Byrne J, Heavey C, Byrne PJ. A review of Web-based simulation and supporting tools. Simulation Modelling Practice and Theory. 2010; 18(3):253–276. https://doi.org/10.1016/j.simpat.2009.09.013

72. Rossberg A. WebAssembly Core Specification;. Available from: https://www.w3.org/TR/wasm-core-1/.

73. Klemenschits X, Manstetten P, Filipovic L, Selberherr S. Process Simulation in the Browser: Porting ViennaTS using WebAssembly. In: 2019 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD). IEEE; 2019. p. 1–4. Available from: http://dx.doi.org/10.1109/SISPAD.2019.8870374.

74. Ligon TS, Leonhardt C, Rädler JO. Multi-Level Kinetic Model of mRNA Delivery via Transfection of Lipoplexes. PLOS ONE. 2014; 9(9):e107148. https://doi.org/10.1371/journal.pone.0107148 PMID: 25237886

75. Leonhardt C, Schwake G, Stögbauer TR, Rappl S, Kuhr JT, Ligon TS, et al. Single-cell mRNA transfection studies: Delivery, kinetics and statistics by numbers. Nanomedicine: Nanotechnology, Biology and Medicine. 2014; 10(4):679–688. https://doi.org/10.1016/j.nano.2013.11.008 PMID: 24333584

76. Phillips A, Cardelli L. Efficient, Correct Simulation of Biological Processes in the Stochastic Pi-calculus. In: Calder M, Gilmore S, editors. Computational Methods in Systems Biology. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer; 2007. p. 184–199.

77. Priami C. Stochastic π-calculus. The Computer Journal. 1995; 38(7):578–589. https://doi.org/10.1093/comjnl/38.7.578

78. Helms T, Ewald R, Rybacki S, Uhrmacher AM. Automatic Runtime Adaptation for Component-Based Simulation Algorithms. 2015; 26(1). https://doi.org/10.1145/2821509

79. Ostrenko O, Incardona P, Ramaswamy R, Brusch L, Sbalzarini IF. pSSAlib: The partial-propensity stochastic chemical network simulator. PLOS Computational Biology. 2017; 13(12):e1005865. https://doi.org/10.1371/journal.pcbi.1005865 PMID: 29206229

80. Cao Y, Gillespie DT, Petzold LR. Efficient step size selection for the tau-leaping simulation method. The Journal of Chemical Physics. 2006; 124(4). https://doi.org/10.1063/1.2159468 PMID: 16460151

[F] A fast embedded language for continuous-time agent-based simulation

In agent-based simulation methods and applications, discrete timestep approaches prevail. To support continuous-time agent-based simulation, we analyze how methods for simulating population-based Con-tinu-ous-Time Markov Chains (CMTCs) can be adopted and derive implications for the concrete realization. To corroborate our findings, we develop an efficient internal domain-specific language (DSL) based on ML3, a modeling language for linked lives in demography. The design as an internal DSL, implemented within the Rust programming language, allows the modeler to exploit the complete feature set of the host language, such as data types and structures, when programming decision processes. A concise and expressive modeling of an agent's discrete decisions and behavior introducing exponentially distributed sojourn times can be supported by adapting the concept of guarded commands from population-based CTMCs.

The execution of models relies on an optimized version of the direct method. This method is a variant of stochastic simulation algorithms, an established method for executing population-based CTMCs in other application areas, notably biochemistry. To efficiently handle the large set of possible transitions inherent to continuous-time agent-based models, we use a dependency graph whose updating scheme caters to the dynamic dependencies within agent-based models and the need for efficient implementation. The presented case studies include implementations of a continuous-time, agent-based migration model and a comparative performance study based on an extended SIR model of infection spread, allowing us to draw conclusions about the impact of different design choices on efficiency.

[274] T. Köster, O. Reinhardt, M. Hinsch, J. Bijak, and A. M. Uhrmacher, *A Fast Embedded Language for Continuous-Time Agent-Based Simulation*, Journal of Artificial Societies and Social Simulation **27**, 10 (2024)

JASSS
THE JOURNAL OF ARTIFICIAL SOCIETIES AND SOCIAL SIMULATION

A Fast Embedded Language for Continuous-Time Agent-Based Simulation

Till Köster[1], Oliver Reinhardt[1], Martin Hinsch[2], Jakub Bijak[3], Adelinde M. Uhrmacher[1]

[1] *University of Rostock, Institute for Visual and Analytic Computing*
[2] *University of Glasgow, MRC/CSO Social and Public Health Sciences Unit*
[3] *University of Southampton, Department of Social Statistics and Demography*
Correspondence should be addressed to *till.koester@uni-rostock.de*

Journal of Artificial Societies and Social Simulation 27(1) 10, 2024
Doi: 10.18564/jasss.5232 Url: http://jasss.soc.surrey.ac.uk/27/1/10.html

Received: 03-01-2023 Accepted: 01-11-2023 Published: 31-01-2024

Abstract: In agent-based simulation methods and applications, discrete timestep approaches prevail. To support continuous-time agent-based simulation, we analyze how methods for simulating population-based Continuous-Time Markov Chains (CMTCs) can be adopted and derive implications for the concrete realization. To corroborate our findings, we develop an efficient internal domain-specific language (DSL) based on ML3, a modeling language for linked lives in demography. The design as an internal DSL, implemented within the Rust programming language, allows the modeler to exploit the complete feature set of the host language, such as data types and structures, when programming decision processes. A concise and expressive modeling of an agent's discrete decisions and behavior introducing exponentially distributed sojourn times can be supported by adapting the concept of guarded commands from population-based CTMCs. The execution of models relies on an optimized version of the direct method. This method is a variant of stochastic simulation algorithms, an established method for executing population-based CTMCs in other application areas, notably biochemistry. To efficiently handle the large set of possible transitions inherent to continuous-time agent-based models, we use a dependency graph whose updating scheme caters to the dynamic dependencies within agent-based models and the need for efficient implementation. The presented case studies include implementations of a continuous-time, agent-based migration model and a comparative performance study based on an extended SIR model of infection spread, allowing us to draw conclusions about the impact of different design choices on efficiency.

Keywords: Domain-Specific Language, Population-Based Models, Agent-Based Models, Continuous-Time Markov Chains, Simulation, Performance

Introduction

1.1 Agent-based modeling has become an established approach for simulating and analyzing complex social systems (Gilbert & Troitzsch 2005; Bianchi & Squazzoni 2015; Macal 2016; Keuschnigg et al. 2018). The challenges of developing and applying agent-based models pervade all steps in conducting a simulation study, including how to program or specify the models, as well as how to calibrate, verify, validate, share and document them (Manson et al. 2020). Problems typically encountered throughout a simulation study appear aggravated due to the nature of agent-based models, their flexibility, and their prospect of capturing multi-level, dynamic phenomena easily and intuitively (Bonabeau 2002).

1.2 These methodological challenges are answered by a plethora of agent-based modeling and simulation tools (Abar et al. 2017). The main criteria to assess and compare tools are ease of modeling and scalability or, closely related, the efficiency of execution. To ease modeling, agent-based modeling tools, such as Mason, Repast, or NetLogo, provide domain-specific languages (DSLs), standard application programming interfaces (APIs), agent

templates, or libraries of procedures. In contrast to general-purpose languages, domain-specific languages focus on particular application domains. External and internal (or *embedded*) domain-specific languages are distinguished. An external DSL is an independent language with its own syntax and semantics, which is usually parsed and executed using a general-purpose language. Internal DSLs, in turn, although appearing as independent languages, are implemented as APIs within a general-purpose language (Fowler 2010). They provide the full expressiveness of the host language and its features, such as inheritance or type systems. These features are helpful if more complex agent states and behaviors, such as those required for BDI (Belief—Desire–Intention) agents (Caillou et al. 2017), need to be modeled. However, unlike in external domain-specific languages, accessing the model structure for analysis or a more efficient simulation is less straightforward.

1.3 Conducting agent-based simulations can be computationally challenging. To increase scalability and efficiency, simulation engines may exploit parallelism (Collier & North 2013), rely on approximate calculations (Wilsdorf et al. 2019; Niemann et al. 2021), or on selecting and configuring simulation algorithms on demand (Helms et al. 2015). Thereby, model structure and properties play an essential role in partitioning the model (Cordasco et al. 2018) or selecting a suitable simulation algorithm (Helms et al. 2015).

1.4 The default execution scheme of agent-based modeling and simulation tools relies on fixed-increment time advances (Abar et al. 2017) and, consequently, the dynamics of "virtually all" (Law 2015, p.704) agent-based models define and use discrete timesteps. The discrete-time approach might threaten the validity of simulation results (Buss & Al Rowaei 2010; Law 2015, pp.72f.) and require specific care in selecting time steps (Köster et al. 2020) and scheduling strategies for agents' interaction (Weimer et al. 2019; Özmen et al. 2016). An alternative approach, *discrete event* simulation, which uses continuous time, circumvents this problem and, in some instances, might be more suitable for the simulation study (Willekens 2017; Niemann et al. 2021). However, providing efficient discrete event simulators of large continuous-time agent-based models faces specific challenges and often requires specific solutions (Andelfinger & Uhrmacher 2021).

1.5 One class of continuous-time agent-based models is population-based continuous-time Markov chains (CTMCs), where all waiting times between discrete events are exponentially distributed. A wide variety of stochastic discrete event simulation algorithms (SSAs) have been developed for these models. However, these aim to model macro-level populations rather than individual agents and are tuned to the requirements of their major application field of biochemical models (Gillespie 2007; Schnoerr et al. 2017). In this area, the modeling needs are met by a series of external domain-specific modeling languages, which are rule-based such as BioNetGen (Harris et al. 2016) or Kappa (Danos & Laneve 2004) or graphical such as Funahashi et al. (2008). The clearly defined and restricted scope of models also allows standardized formats for automatically exchanging models between simulation tools (Keating et al. 2020). The individuals of the population are typically simple agents with only a few attributes with a finite and small value domain, e.g., a protein being phosphorylated or not. Thus, generally, large groups of "identical" individuals exist that can be handled as sub-populations. Together these sub-populations form the current state of the model. The dynamics of these biochemical systems boil down to the binding, creation, or removal of agents. The simple structure of the models means that they can also be transformed into ordinary differential equations (Mendes et al. 2009). This is usually a good approximation if populations are large and stochastic effects may be ignored.

1.6 In the following, we will pursue the question of what the modeling and simulation of continuous-time agents in sociology and demography can learn from experiences in the application field of biochemistry. We will revisit central modeling concepts of the external domain-specific modeling language ML3, which allows succinct modeling of the dynamics of agent-based CTMCs. Identifying central modeling concepts of ML3 and the limitations of the external DSL will form the basis for developing an internal DSL in the Rust programming language. This implementation allows the utilization of the host language's features and, at the same time, caters to the specific needs of the modeling domain. Using an internal DSL has implications for the efficient execution of the models. We will discuss this next, comparing the efficient execution of such agent-based CTMCs to population-based CTMCs, focusing on maintaining dependency graphs during execution. Based on this analysis, we present ML3-Rust and the realized simulation algorithm. Finally, we utilize ML3-Rust to model and simulate migration processes and infections. For the latter, we conduct a comparative performance analysis to discuss the efficiency of different languages and modes of implementation.

Modeling Concepts for Continuous-Time Agent-Based Models

2.1 Domain-specific modeling languages reflect the central modeling concepts of the domain, such as reactions for modeling biochemical systems or interacting agents for modeling demographic processes. The motivation for

developing the Modeling Language for Linked Lives (ML3) has its roots in social science applications. In ML3, individuals do not exist in isolation, but their *lives* are *linked* through social networks or other connections. This observation has led to the need to succinctly describe the diverse decision processes of such linked lives in continuous time (Warnke et al. 2017). The context for individuals to make decisions is formed by their social network. Further requirements are behavior conditional on agent attributes, such as age-dependent behavior and stochastic waiting times (Reinhardt et al. 2021). Alternative decisions can be modeled as concurrent processes that compete by stochastic race (Warnke et al. 2017).

2.2 In ML3, agents represent any entity in the model, including individual persons, but also higher-level actors such as households, towns, governments, or other policy-makers. Each agent has a **type** α, and the type determines its attributes and behavior.

2.3 The relations between agents are represented with **links**. Links are specified between agents of a specific type. They might refer to social ties, linking an agent to friends or an agent being part of a household. Links also allow the introduction of discrete locations, for example linking agents to the town where they are located. The definition of agents in ML3 is similar to other agent-based modeling and simulation tools, neither less nor more succinctly. The notion of time advances whether in discrete timesteps or after sojourn times sampled from a continuous distribution makes no difference to how the agents, their properties, and their interactions are modelled.

2.4 Consequently, the distinguishing language concept in comparison to the timestepped approaches and the core of each model in ML3 are its stochastic **rules**, which describe the agent's behavior in continuous time. The rules in ML3 are inspired by guarded commands. They consist of three parts: *guard*, *rate*, and *update*. This triple has been found to form a natural description for continuous-time population-based models (Henzinger et al. 2011). Each rule in ML3 is assigned to a specific type of agent and specifies under which condition, at which rate, and which state changes occur, in general form:

$$\alpha : c \xrightarrow{r} e \tag{1}$$

2.5 The rule can be applied to agents of type α. The **guard** condition c evaluates to a boolean variable, further constraining the rule's application. The **effect** e specifies the state change when the rule is executed. An agent's transition can affect its own state but also the state of other agents as well as links of its social network. Finally, the **rate** r, an expression that evaluates to a real number (plus a symbol for infinity to allow rules to apply instantaneously), specifies the timing of the behavior.

2.6 The following simple rule is inspired by a network-based SIR (susceptible-infected-removed) model (Kermack & McKendrick 1927). An agent which is susceptible (condition c) becomes infected (effect e) at a rate r that is calculated by multiplying a rate constant with the number of infected neighbors:

$$Person : status = susceptible \xrightarrow{\beta \cdot \#\text{ of infectious neighbors}} status := infected \tag{2}$$

2.7 The SIR model can also be used to illustrate the transition from population-based CTMCs to agent-based CTMCs. In the basic SIR model, all susceptible, infectious, and recovered individuals can be grouped together, and the state consists of the number of members in each group. This population-based approach still works if a few age classes and a few discrete locations are added as attributes to the agents. However, once each individual's social network influences the chance of infection, agents can no longer be easily aggregated and thus are treated individually.

2.8 The rules define the continuous-time (aka discrete-event) dynamics of the model. During the simulation, each pair of an agent and a matching rule yields an **instance** of the rule. Intuitively, for each instance, a random waiting time is drawn from an exponential distribution parameterized with the value of the rate r. The instance with the minimal waiting time is selected and executed, and the time is advanced by that waiting time. Formally, this execution yields a Continuous-time Markov Chain (see e.g. Reinhardt et al. 2021).

2.9 It should be noted that ML3 also supports time-dependent (or age-dependent) rates and the scheduling of events at arbitrary times. These features become important if activities are age-dependent (such as mobility) or events such as retirement occur at a specific time in an agent's life course (Warnke et al. 2017). Therefore the semantics of fully-fledged ML3 models is that of Generalized Semi-Markov Processes (Reinhardt et al. 2021). However, here, this is of less importance.

2.10 So far, we have defined the syntax of an ML3 rule in an abstract manner. Figure 1 shows the SIR rule implemented in the concrete syntax as supported by the external domain-specific language of ML3. The first version of ML3 has been realized as an external DSL (Warnke et al. 2017). The elements of Equation (2) can be easily identified in the code.

```
Person
| ego.status = "susceptible"
@ beta * ego.neighbors.filter(alter.status = "infectious").size()
-> ego.status = "infectious"
```

Figure 1: Example of concrete external DSL syntax.

2.11 The external DSL ML3, including its simulator, has been successfully applied to different case studies of demographic migrations (e.g. Warnke et al. 2017; Reinhardt et al. 2019). In addition, a macroeconomic model to study monetary regime shifts (Peters et al. 2022) and a model to study the impact of sanctioning on recreational fisheries (Haase et al. 2022) document the versatility of ML3. However, if agents require more complex belief structures than simple sets, these need to be modeled by defining a separate agent object to contain and capture the beliefs, and subsequently linking them to the original agents. Whereas this modeling of a single agent as a set of interacting agents mimics a component-based design of agents as adopted, for example, in some software agent architectures (Müller & Pischel 1993), it might not reflect the perception of the modeler, and as a result, hamper a straightforward model design. However, to support flexible and safe modeling in an external domain-specific language, data structures, type systems, and other features of general-purpose languages have to be re-implemented, which led us to reconsider our design choice to implement ML3 as external DSL.

2.12 In contrast to external DSLs, internal DSLs allow users to exploit the full feature set of general-purpose programming languages in modeling agents and, thus, widen the scope to more elaborate continuous-time agent models. Not surprisingly, major agent-based modeling and simulation tools rely on internal domain-specific languages with imperative programming styles (albeit, as stated above, typically with discrete stepwise semantics). For example, Repast Simphony is a well-known example of an internal DSL or API for agent-based modeling (North et al. 2013). Repast uses Java as its host language. In contrast, the comparatively simple, well-defined structure of agents and their dynamics enabled the successful design and establishment of small, declarative, external domain-specific languages for modeling population-based CTMCs in biochemistry, such as BioNetGen (Harris et al. 2016).

2.13 Let us take a closer look at why there could be a mismatch between external domain-specific modeling languages and agent-based modeling (in general). NetLogo is a multi-agent programming language and modeling environment implemented in Java and Scala, which is widely used across various application domains. The offered modeling language is NetLogo. Therefore, one might feel inclined to call NetLogo an external domain-specific modeling language. However, DSLs are dedicated, small languages focusing on particular aspects of software systems (Fowler 2010). NetLogo is a programming language derived from Logo, to which it adds agents and concurrency (Tisue & Wilensky 2004). Logo has been designed as a general-purpose programming language to be usable by children. The latter caters to the first principle of NetLogo, namely its *low threshold* in getting started, and the former to the second principle, i.e., *no ceiling*: that the language should not constrain advanced users in their programming. This flexibility to program agents every way the modeler likes appears as a central and distinctive requirement (Bonabeau 2002) and, consequently, best supported (if the class of agents is not constrained) by embedding the DSL in a general-purpose language. Similar efforts can be found in other communities, e.g., the `Symbolics.jl` library where a Computer Algebra System is implemented within the Julia language (Gowda et al. 2022).

2.14 In the next section, we examine the possible implications of internal DSLs on designing efficient simulation algorithms for continuous-time agent-based models (compared to continuous-time population-based models).

Model Execution: Algorithms for Simulating Agent-based CTMCs

3.1 In continuous-time agent-based CTMCs, each agent in the system may undergo one of several transitions. Each of these transitions has an associated *rate*. This rate is computed by the rate function and corresponds to the likelihood of this transition occurring in a specific time interval. The standard approach of discrete event simulation is via scheduling based on event queues (Law 2015). For each potential transition, in the case of CTMCs, delay duration needs to be drawn from an exponential distribution. Based on the current time plus the calculated delay, a time stamp is assigned to the transition and it is stored in the event queue. The basic discrete event simulation algorithm takes the event with the smallest time stamp that is scheduled and processes it. This procedure will usually lead to new events being scheduled or the conditions for previously scheduled events

not being met anymore. Those events are either removed from the event queue or rescheduled. Due to the inherent lack of memory of the exponential distribution, the delay may be arbitrarily redrawn. Please note we will omit time-dependent rates for now, as they require specific treatment (Reinhardt et al. 2021).

3.2 This procedure is similar to techniques of Stochastic Simulation Algorithms (SSA) that are applied to population-based CTMCs and have become particularly popular in the field of biochemical reaction models. In this field, various simulation algorithms and tools exist (for example, Danos & Laneve 2004; Harris et al. 2016; Schnoerr et al. 2017). A review of the standard SSAs (also called Doob-Gillespie simulation algorithms) can be found in Gillespie (2007).

3.3 The so-called *Direct Method* is one possible formulation for solving the scheduling problem (Figures 2a and 2b). Instead of calculating times points for transitions, it operates directly on the propensities (Gillespie 1977, 1976). This by itself is not necessarily an advantage but allows for many further optimizations (Schnoerr et al. 2017). In the Direct Method, at every step of the propagation, for all transitions in the system, we calculate their respective propensity. We can then pick both a time step and the reaction. The reaction to be executed is selected via a weighted random choice algorithm with the individual propensities as weights. The time interval (sojourn time) to execute the reaction is exponentially distributed and relates the reaction's propensity to the sum of all propensities. The equivalent SSA to event scheduling in continuous time, as introduced above in the context of CTMCs, is called the Next Reaction Method (Figure 2c).

```
update_step():
  reac = select_weighted_random(reaction_rates)
  execute(reac)
  for r in all_reactions:
    update_rate(r)
```

(a) In the *Direct Method*, the reaction is selected by weighted random choice. After a reaction is executed, all rates are recalculated.

```
update_step():
  reac = select_weighted_random(reaction_rates)
  execute(reac)
  for r in dependent_reactions(reac):
    update_rate(r)
```

(b) In the *Dependency Graph Direct Method*, the reaction is selected by weighted random choice. Rate updates are done following a dependency graph.

```
update_step():
  reac = top(queue)
  execute(reac)
  for r in dependent_reactions(reac):
    reschedule(r)
```

(c) In the *Next Reaction Method*, a queue is used to find the next reaction via scheduling. To only reschedule reactions that were altered, a dependency graph is used.

Figure 2: Simplified pseudo-codes of different algorithms.

3.4 To pursue the question of whether and how insights from SSA and the developed algorithms apply to simulating continuous-time agent-based models, we discuss the differences and similarities in simulating population-based CTMCs. We base this on the example of biochemical models and continuous-time agent-based models based on the example of ML3 (see Table 1).

	population-based models (biochemical)	agent-based CTMCs (ML3)
multiplicity	populations	individuals
dependency graph	static	dynamic
state	array of integers	dynamic set of interrelated agents
locality of effects	constrained to decreasing reactants and increasing products of the reaction	unconstrained
scheduling of next event	depending on rate constant and amount of reactants	function of the rate constant and the current state

Table 1: Similarities and differences between simulating population-based models and agent-based CTMCs

3.5 In population-based CTMCs models, the state is usually made up of various sub-populations (integer amounts) of different species. In most agent-based modeling, such as ML3, on the other hand, individuals need to be considered. Thus, instead of counting the amount of a specific species, e.g., 900 receptors being phosphorylated, the simulation algorithm distinguishes between individual agents, e.g., as each migrant has an age, a location, and specific relations to other agents. The larger number and often metrical type of attributes prevent grouping agents into (sub-)populations. This has implications not only in terms of memory usage but also for the number of transitions scheduled in the system.

3.6 A reaction has an aggregated rate (so-called propensity) that takes into account its rate constant and multiplicities of the members of the (sub-)populations. For example, given the reaction of $A + B \rightarrow C$ with rate constant r, there would only be **one** transition, and assuming mass action kinetics, the rate of the transition would be $r \cdot \#A \cdot \#B$, where $\#A$ and $\#B$ are the respective population sizes of species A and B in the current state. In a continuous-time agent-based scenario, possible transitions need to be considered for each individual agent. Therefore, it becomes crucial for the simulation algorithm to efficiently handle large numbers of possible transitions.

3.7 The rate function of a biochemical reaction depends on the rate constant and the current amount of its reactants. In the standard Direct Method, every propensity is recalculated at every step. This recalculation is inefficient as most propensities do not change when one transition of an agent-based model is executed. The same holds for large biochemical networks. Therefore, several algorithms for biochemical reaction networks use dependency tracking (Gibson & Bruck 2000). Here propensities are only recalculated where needed. This is achieved by having a dependency graph that connects reactions (Figure 2b). This graph is used to identify all transitions that depend on that value whenever a change is made. Only for these transitions, the rate calculation is repeated.

3.8 In agent-based models, the dependency structure of a transition is typically dynamic, not only depending on an agent's attributes, but its relations to other agents at that point in time and the state of those related agents (e.g., the health status of neighbors in the SIR network model). These dynamic interaction structures are considered a defining characteristic of agent-based models (Uhrmacher et al. 2000). Thus, in comparison to biochemical models, continuous-time agent-based models, such as those in ML3, add an additional layer of technical complexity to these algorithms, as the dependency graph is dynamic and, consequently, has to be frequently recalculated.

3.9 When selecting a reaction, we need to perform a weighted random choice on the propensities. This can be time intensive if done in a basic linear fashion. Therefore optimized approaches, such as static (Cao et al. 2004) or dynamic (McCollum et al. 2006) sorting, and more advanced data structures exist. Storing propensities in a tree allows for changes in propensity values and selection with logarithmic complexity (Köster & Uhrmacher 2018).

3.10 The effects of a biochemical reaction are constrained to decreasing the sub-populations of its reactants and increasing the sub-populations of its products. On the other hand, the effects of transitions of agent-based models are not constrained to the agent locally but might affect the environment, the existence of agents, or their links. For example, if the parents decide to move to a different location for underaged children, this also typically implies a change in their location. This tight coupling of agents provides additional challenges for synchronization, e.g., to allow an efficient, parallel execution (Andelfinger & Uhrmacher 2021), and for accessing the state of the model.

3.11 Rates and effects in ML3 are expressed as arbitrary functions accessing the agents, their states and network. Thus in external DSLs complex expressions need to be parsed and evaluated by the simulator. To efficiently handle these expressions would require an optimized compiler for the external DSL. While some tooling for such an optimization exists (see, for example, Lattner & Adve 2004), it is not yet always easy to use. Alternatively, one may leverage existing language infrastructure through code generation. Here the model code is read, and the source code for an existing general-purpose programming language is automatically generated. Meyer et al. (2018) and Köster et al. (2022) show efficiency improvements through both run-time and ahead-of-time code generation. Further details of this trade-off have been discussed in Warnke (2021), Barringer & Havelund (2011), and Artho et al. (2015). For some external DSLs evaluating complex expressions remains a performance challenge. However, in many external DSLs for population-based modeling and simulation, this is not as much of a problem, as rate expressions and effects are constrained. In the context of continuous-time agent-based modeling, this provides another argument for an internal rather than external modeling language for continuous-time agent-based models.

Implementing ML3 as an Embedded, Internal DSL in Rust

4.1 Developing an embedded or internal DSL implies providing a set of abstractions in a framework written in a general-purpose language. We can then rely on all the existing capabilities of that host language. These include using arbitrary data structures and efficient compilation of model code. Given the requirements of continuous-time agent-based models discussed above, an embedded approach appears best suited to broaden the scope of models ML3 can handle. As host language, we use `Rust` (Matsakis & Klock II 2014), a modern programming language for reliable and efficient software.

4.2 The core challenge of implementing ML3 as an internal DSL in Rust is, on the one hand, to provide a similar abstraction and the possibility of succinct model specification similar to the original external modeling language ML3 and, on the other hand, to support efficient execution of these models. For the latter challenge, tracking the dependencies needed for selective/efficient propensity updates is crucial, also to keep in line with the requirements of the host language.

4.3 For efficient execution, the main challenge is tracking attribute writes and reads to update the dynamic dependencies. An implementation as an external language has a clear advantage since the propensity and update functions are known. The syntax tree is evaluated at every step by the simulator. In this process, dependencies and changes in the dependency graph can easily be tracked (Reinhardt & Uhrmacher 2017). At the same time, dependencies cannot be as easily tracked in internal DSLs. Previous solutions to this problem included using a feature within the java virtual machine where it is relatively easy to define cut points within the program (in our case, the read and writes) and add specific code (in our case, updates to the dependency graph) (Kreikemeyer et al. 2021). This is also a type of code generation. This is a top-down approach, where the update code is added later on top of the model.

4.4 For our implementation, we have chosen a different approach more suited to the workflow of ahead-of-time compiled languages which allows some further optimizations. The user specifies the structure of the agents and their interaction channels (i.e., the edge types) in the system. We can then process these definitions using Rust's built-in macro system and generate getters and setters. These typed getters and setters are the only public interface to the agents' attributes. They also can be used to retrieve agents. In this bottom-up approach, the modeler builds the model on top of the interface. An example of a rule defined in the internal domain-specific language of ML3 can be found in Figure 3. Transitions are imperatively added as a rule triple made up of lambda functions.

4.5 Most transition rates in ML3 are constant in time and only change when the attributes and states (as captured by the dependency graph) change. However, some transitions are either scheduled (i.e., happen at a specific time) or are time-dependent. The first is relatively easy to handle. In addition to the rate-based handling for our dependency graph Direct Method, we manage an event queue. When a transition is executed, it is checked whether a queued transition is next. Time-dependent reactions are more of a challenge (Reinhardt et al. 2021). The user specifies a timestep that is "small enough" to capture the characteristic changes of the propensity expression. The propensity is then regularly recalculated with this timestep. This is an approximation, but only to the degree that the rate changes quicker than the selected timestep. This has two advantages compared to earlier realizations (Reinhardt & Uhrmacher 2017). Firstly, no propensity calculation is wasted. Secondly, it integrates better with the existing rate or propensity-based transitions.

4.6 All agents are stored as typed variants in a contiguous array. This is not optimally memory efficient, as each agent takes as much space as the largest possible agent. However, iteration over agents and de-referencing

```
model.add_transition_for_Person(
  /* guard */ |ego| ego.get_status() == HealthState::Susceptible,
  /* rate */
  |ego| {
    beta * ego.network()
      .filter(|alter| alter.get_status() == HealthState::Infected)
      .count() as f64
  },
  /* effect */ |ego| ego.set_status(HealthState::Infected),
);
```

Figure 3: Example of concrete internal DSL syntax for an infection process. This is the same rule as the one shown in Figure 1

via IDs/indices is faster than in a typical pointer-based (dispersed) map or set data structure. All attributes are individually tracked via a generated index. The agent types are described as structured *enums* (variables of the enumerated type). Based on this, several interfaces are generated. The most important one is a wrapper class that allows both mutable and immutable access to the attributes of the agent class via functions. Internally, this wrapper consists of the ID of the agent and a reference to the simulator state. Whenever access is made, the required value is looked up in the state, and the appropriate changes to the dependency graph are made. Further generated functionalities include functions for adding and removing agents of a particular type as well as edges. Edges are also typed, providing compile type correctness checking for compatibility. Some basic facilities to compute observables are also included, like counting agents with a particular attribute expression.

4.7 An interesting optimization is that of so-called *lazy dependency graph updates*. Each change in the dependency graph is relatively expensive. In our experience, for many models, the graph remains relatively constant. That is, even after an expression needs reevaluation, it still depends on the same attributes of the same agents. While this is not always the case, it can still be exploited as an optimization. Instead of clearing the entire graph and rebuilding it whenever an expression is computed, we only check if the new dependencies are already present. If need be, the dependencies can still be added. However, there is no deletion of unused dependencies. Periodically, using a counter, all dependencies are reset to avoid overflow.

4.8 Expressing experiment design and related workflows concisely yet expressively is a topic of ongoing research (Kleijnen 2018). The previous external language version of ML3 was integrated with SESSL, an embedded domain-specific language for specifying simulation experiments (Ewald & Uhrmacher 2014). A final workflow for Rust implementation still needs to be established for setting up experiments. In the past, we have used external Python scripts and experiments integrated with the model formulation in Rust. The realization as an embedded language also enables using this languages ecosystem like the `egobox` framework for optimization experiments (Lafage 2022).

Case study 1: Implementing a Complex Model of Migrant Routes

5.1 As one case study to illustrate the properties of the ML3-Rust implementation, with an individual-level agent-based model with a relatively complex agent structure, we present a simplified version of a model of the effect of information exchange on the emergence and changes of migration routes (Hinsch & Bijak 2022). Originally this model was implemented in ML3 alongside a general-purpose language realization in the Julia language (Reinhardt et al. 2019; Bijak et al. 2021). The model has been motivated by real-life policy challenges generated by the emergence and re-emergence of migrant routes in Europe. These migration processes had amplified since the 2010s, especially in the context of the civil war in Syria. We give a brief overview of the model in the following paragraphs. For more detailed information, we would like to refer the reader to the publications on the original model and the ML3 version (Reinhardt et al. 2019; Bijak et al. 2021; Hinsch & Bijak 2022). The model, simulator implementations, and scripts to execute the experiments and plot the data are available at: `https://git.informatik.uni-rostock.de/mosi/ML3-Rust`.

Model overview

5.2 In the model, agents attempt to migrate across a world consisting of cities connected by transport links. Agents start with limited or no knowledge about the world but have to obtain information by exploring their surroundings or communicating with other agents. Information is generally unreliable, as exploration, as well as communication, can be imperfect or error-prone. We investigate how the optimality and predictability of migration routes depend on the degree to which agents rely on their peers for information.

5.3 The world consists of a random graph of cities and transport links (see Figure 4). Cities have quality and resource availability, where higher values make them more attractive to agents. A small number of cities function as entries and exits, respectively. Transport links connect two cities and have a friction value representing ease of travel, affecting the agents' travel speed.

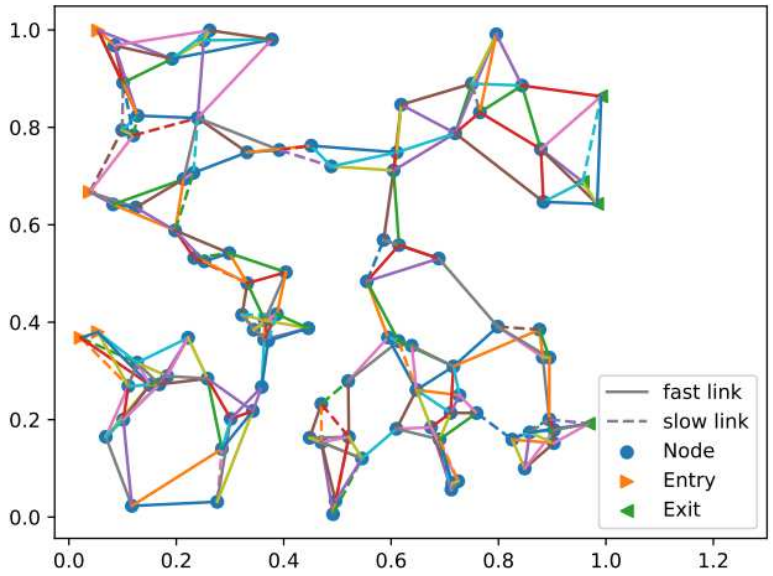


Figure 4: A randomly generated spatial network. The agents travel from Entry to Exit across fast and slow links.

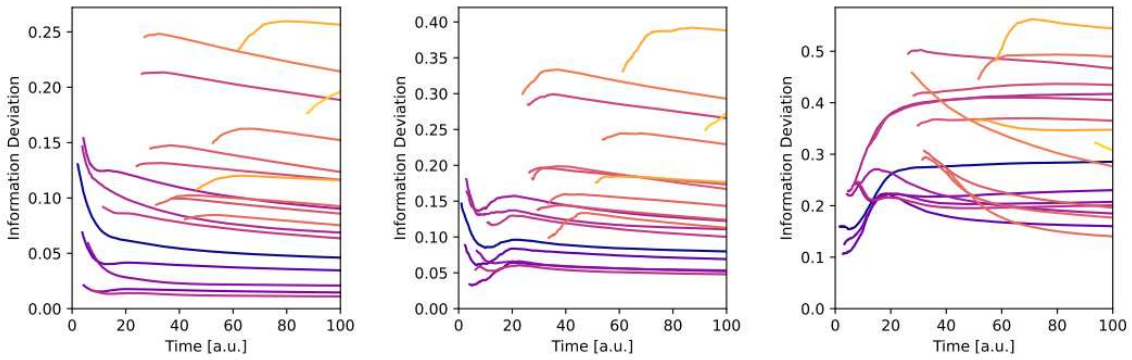
Agents, communication, and information

5.4 Agents start at randomly selected entry cities and attempt to travel to any exit. They decide where to travel based on the information available to them by picking a neighboring city with the best combination of travel time, quality, availability of resources, and proximity to the exit.

5.5 Agents maintain a list of contact agents that they communicate with regularly. Agents can add each other as contacts if they spend time in the same city. Agents also explore the city they are currently staying at, improving their knowledge about the cities' properties and discovering transport links to neighboring cities.

5.6 Each agent maintains an internal (possibly incomplete) model of the world. While topological information (if existent) is always correct, information on the properties of cities and links can be inaccurate: each property (e.g., quality of a city) is represented in an agent's internal model by two numbers – an estimated value and the level of certainty that that value is correct. When exploring, agents' information always improves, i.e., their estimates become more accurate, and their certainty increases. When communicating with other agents, topological information is always transmitted faithfully. Information on properties of cities and links, however, carries a transmission error.

5.7 Furthermore, the effect of perceiving another agent's opinion depends on the combination of the certainty values and the similarity of the estimate of the two agents. In general, agents adapt their estimates based on the other agents' estimates. Still, the agent with higher certainty will convince the one with the lower certainty to change its estimate more. If estimates are similar, both agents will increase their certainty, whereas different estimates can lead to a reduction in certainty for both agents.



(a) Low error in transmission (b) Medium error in transmission (c) High error in transmission

Figure 5: The deviation of the average knowledge about different locations across agents through time for different errors of transmission. The results are averaged across 500 replications. Simulation experiments of this scale were not possible with the previous version of ML3.

Selected results and performance characteristics of the model

5.8 In Figure 5, we show sample results for the deviation of the average knowledge of different locations through time. Each line represents one location. The color of the line corresponds to how far along the x-axis network the node is located. The network used for the simulations is shown in Figure 4. The three subpanels of Figure 5 show how differently the error behaves depending on the error that occurs when transmitting the information. The lower the y-value of the lines in the figure, the "more correct" the total global knowledge of a city is. We observe that for the low error in transmission, the quality of information is very good for the early cities in the migration process. This is because many migrants pass through there and therefore, a lot of high-quality information is available early on. Since the error in transmission is low, the quality stays constant or even slightly increases throughout the simulation. This is different for the model with the high error in transmission. Here we observe a decrease in the quality over time. An initial decrease in the quality is observed for all models for the later locations in the system. Since only a few agents have reached them, the information is spread chiefly via transmission and therefore is more susceptible to errors in transmission. Finally, for the medium error in transmission, we observe an intermediate behavior. First, the information deviation decreases for the starting locations, but then increases as more agents from different starting points come into contact.

5.9 To understand the simulator's performance, we conducted a profiled run of the simulation model. We observed the number of times a particular transition was executed and the duration each execution took. The results are shown in Figure 6. We observe that the absolute transition counts (Figure 6a) are relatively evenly distributed among five different transitions, with the MINGLE transition slightly dominating. However, in the time profiling (Figure 6b), we observe that this is a costly transition and therefore dominates the runtime.

5.10 The most expensive transition (Figure 6c) is communication. However, this transition happens only comparatively rarely and is therefore not dominant for overall performance. However, this transition becomes more expensive over time due to the steadily increasing size of the contact network of agents that have arrived at their location. To achieve longer runtimes with this model, an improvement to this communication mechanism is needed.

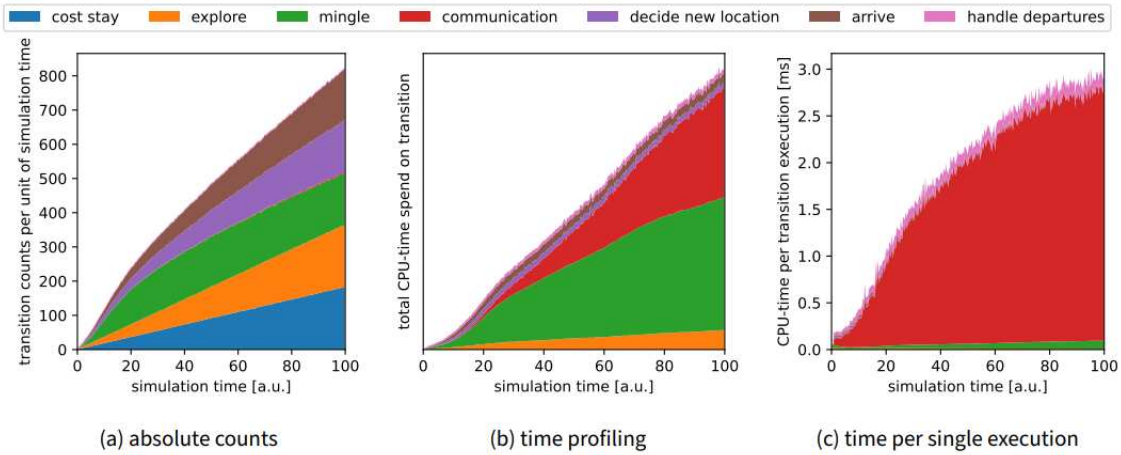


Figure 6: Selected runtime performance metrics for the model of migration route formation. The results are averaged across 500 replications. The wall clock time needed to execute these transitions is shown, not the virtual time within the simulation.

Case study 2: Performance Comparison for a Model of Infection Spread

6.1 In this case study, we discuss the performance characteristics of the ML3 discrete event approach in general and our implementation in particular. We have extended the basic SIR model by placing agents on a spatial grid to analyze the performance of the implementation. SIR has been chosen as a well-known population-based model, a point of reference in epidemiology that is easy to describe, implement and analyze. We compare the performance of ML3-Rust with a few other approaches, including the implementation of ML3 as an external language in Java (Reinhardt et al. 2021). We also wrote custom simulators specifically for the model to better understand the maximum achievable performance for a discrete event implementation. They exploit the regularity in the network by direct index-based access. Additionally, we use our prior knowledge of how the three potential transitions interact. The custom simulator works only for this exact model, but it is exact. We implemented two versions of the custom simulator, one realizes the traditional linear Direct Method still commonly used in cell biology (as shown in Figure 2), and the other manages the propensities within a tree (Köster & Uhrmacher 2018).

6.2 To highlight the unique features of the implementation presented in this paper (high expressive performance with exact continuous-time semantics), we also implemented a discrete timestep version of the model in NetLogo (Wilensky 1999). Whether the transition will occur is randomly sampled at each step. This is a naive (not high-performing) but common way to implement this type of model stepwise. For optimal performance in NetLogo, some considerations have to be made (Railsback et al. 2017), but generally, it is considered to have reasonable throughput compared to other tools (Abar et al. 2017). The NetLogo time extension provides the ability to schedule events in continuous time (Sheppard et al. 2016). However, events can not be canceled or retracted. Therefore there is no means of expressing a stochastic race. Dynamic event scheduling is at the core of most ML3 models, and therefore, an equivalent formulation for ML3 models in NetLogo is not easily accessible. The NetLogo model, custom simulator implementations, and scripts to execute the experiments and plot the data are available at `https://git.informatik.uni-rostock.de/mosi/ML3-Rust`.

Model overview

6.3 The model is a variant of the standard SIR model (Kermack & McKendrick 1927). The agents are laid out in a two-dimensional non-toroidal (i.e., finite) grid with a Von-Neuman 4-Neighborhood. The agents can be in 3 states, susceptible (S), infected (I), and recovered (R). In the presented version of the model ($SIRS$), we added a wearing-off of the immunity to the agents, such that recovered agents turn susceptible again. The exact rules

and rates are:

$$S \xrightarrow{\sum_{I \in \text{Neuman}} 5 \cdot 10^{-2} + 2 \cdot 10^{-6}} I$$

$$I \xrightarrow{10^{-2}} R$$

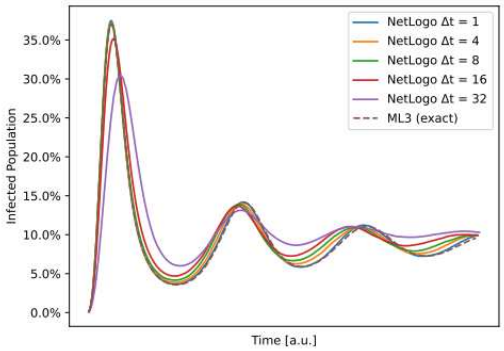
$$R \xrightarrow{1.5 \cdot 10^{-3}} S$$

Initially, 0.1% of agents are infected, and the rest are susceptible.

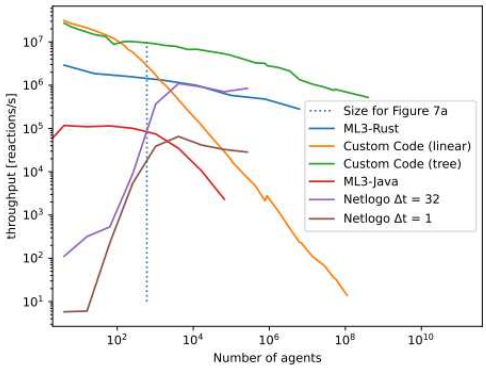
Selected results and performance characteristics of the model

6.4 The model parameterization introduced above leads to a damped oscillation in the number of infected agents. The outputs from the NetLogo implementation and ML3 are shown in Figure 7a. The dotted line shows the exact result of this model in continuous time with exponentially distributed event delays. At 10,000 replications, these results are fully converged. The results are identical for ML3, ML3-Rust, and the two custom implementations of the model, as they all follow the same CTMC semantics. The discrete stepwise model in NetLogo converges to these results when decreasing the timestep.

6.5 In Figure 7b, we show the performance of the different simulators. On the y-axis, we have the throughput of performed transitions per second of wallclock time. This forms a more meaningful and standardized measure of performance than the time to execute a model. The size of the system from Figure 7a experiment is marked with a dotted vertical line (600 agents).



(a) Simulation output for the SIR model with 600 agents. The NetLogo results deviate due to time discretization errors.



(b) SIR performance across different model sizes. Including different versions of ML3 and two custom highly optimized native code implementations.

Figure 7: Results for the second case study, the sir model.

6.6 The measurements for ML3-Rust and the Rust-based custom simulators were done using the Criterion micro-benchmarking framework. This uses advanced statistics such that the overall benchmark execution time is about 5 seconds for each aggregate measurement. For the Java implementation, we measured the total runtime of a number of events (decreasing for larger population sizes to limit total runtime). For NetLogo, we used the headless execution and timed the total runtime of the simulation model. NetLogo does not provide a reactions-per-second metric, as there are no discrete events. We used the duration of the overall simulation run and scaled this with the number of reactions that would have occurred in a discrete event implementation.

6.7 As we are only interested in the steady-state simulation performance, we have therefore subtracted the startup cost from the NetLogo runs by subtracting the cost of a simulation run that only performed a single global update. Multiple replications (depending on the stochasticity) were performed, and the standard error of the mean is shown. For NetLogo, five replications per data point were performed for fewer than 100 agents and three for over 100 agents. The measurements were taken on a Desktop workstation with an Intel Xeon E5-1630 CPU.

6.8 The main result from Figure 7b is the blue line for the ML3-Rust simulator presented in this paper. We observe that the simulator scales well with an increasing number of agents. Note that this does not mean that the total runtime of the simulation is constant across system sizes, only that the time spent per agent is largely independent of the total system size. For the custom simulators, we see the large impact of the used data structure and algorithm for storing the propensities of the system. Whereas for a smaller number of agents, the linear approach is even faster than the tree-based, for larger numbers, we profit from logarithmic scaling when updating or traversing the tree. There is a constant offset (nearly an order of magnitude) between the custom simulator and the rust-based ML3 simulator, indicating that there are inefficiencies in using this generic simulator. Still, overall scaling is not affected by this. When considering large system sizes, the scaling behavior of the algorithms becomes decisive.

6.9 A natural question is whether the custom simulator implementation is genuinely close to the limit of execution speed for this model. One indication of the performance limitations of this simulator is the pronounced shift in simulator throughput at $3 \cdot 10^6$ agents. At this point, the state of the system occupies 48Mb. For each agent, the system's state is comprised of two 64-bit floats corresponding to the agents' propensity and the tree sum at that point and the agents' state, stored as an integer and aligned to 64-bit on our platform. At this point, we are beyond the limit of our CPU's cache memory. If we increase the model size any further, parts of the model will have to be saved in the main memory, which comes with a performance penalty. While this does not guarantee that our implementation is optimal, it indicates that most compute operations could not be optimized much more, as the CPU mostly has to wait for data from memory for larger models. This could potentially be reduced further, but not by much, indicating that this algorithm is implemented at least near the maximum efficiency.

6.10 The NetLogo results show a very large computational overhead: it is by far the slowest for small system sizes. The per-agent performance increases and is steady for more than 1,000 agents. Algorithmically speaking, one would expect that performance will stay constant beyond the initial overhead per agent as each agent is visited once per step. The NetLogo variant, with a huge time step, reaches the performance of ML3-Rust. However, as we have seen above, the simulation outputs show a large deviation for this setting. With a much smaller timestep, the performance naturally deteriorates (roughly by a factor of 32, as that is the difference in the length of the timesteps under consideration). Larger agent populations could not be implemented with NetLogo due to memory limitations. The performance and potential of using explicit discrete event simulation within NetLogo have been addressed via an extension (Railsback et al. 2017).

6.11 The legacy Java-based ML3 implementation shows similar scaling behavior as the custom simulator using linear search. This indicates that even though the used Next Reaction Method should scale the same as a tree-based propensity approach, some part of the current implementation seems to iterate through all agents or reactions at every step. Furthermore, we observe a constant performance difference, likely due to the chosen language (java) and data structures.

Conclusion and Future Work

7.1 In this paper, based on the example of ML3, we have analyzed challenges in developing domain-specific modeling languages for continuous-time agent-based models. ML3 is a domain-specific modeling language aimed at modeling linked lives in demography through continuous-time agent-based models. With ML3-Rust, we have realized an efficient internal domain-specific language.

7.2 In contrast to external domain-specific languages, internal or embedded languages allow the modeler to exploit the full set of features of the host language and the modeler to program the agent as they like. This is of particular interest if more complex data structures and decision processes need to be programmed to govern the agents' behavior. In addition, the modeler can use a general-purpose language they might already be acquainted with. The disadvantage of internal domain-specific languages is that the language's syntax cannot be as freely defined as it is still that of the host language.

7.3 For our implementation, we have adapted stochastic simulation algorithms that are highly popular in the area of population-based CTMCs, in particular biochemistry. Our analysis and the realization of ML3-Rust revealed similarities and differences between simulating populations of discrete entities in biochemistry and large sets of individual agents in agent-based models. The key conclusions from this comparison are as follows:

- For population-based CTMCs, in particular in the biochemical area, external domain-specific modeling languages prevail due to the restricted scope of models defined by biochemical reaction networks. Without such a restricted modeling scope, internal domain-specific languages become a better fit for the desire of modelers to program their agents to suit their needs. In continuous-time agent-based simulation,

agents cannot easily be aggregated into subpopulations resulting in a high number of possible transitions that compete with each other. Therefore efficient handling of transitions becomes paramount.

- When it comes to modeling continuous-time agent-based models, one distinctive part is the description of behavioral rules to support arbitrary sojourn times. Here, the concept of *guarded commands*, with guards, rates, and effects, as proposed for continuous-time population-based models, allows the modeler to specify the agents' dynamics in continuous time succinctly. This simple and concise yet versatile structure can also be realized in an internal domain-specific language. Internal domain-specific languages enable compiler optimizations for handling complex rate expressions (as they are characteristic for continuous-time, agent-based models) efficiently during simulation.
- Stochastic simulation algorithms that maintain dependency graphs and are developed for dealing with large sets of possible biochemical reactions can support equally an efficient simulation of continuous-time agent-based models. Therefore, an internal DSL needs to keep track of these dependencies transparently to the modeler. In addition, suitable means for efficiently updating the dependency graph are crucial since, in contrast to population-based CTMCs, dependency graphs in continuous time agent-based models are dynamic. A *lazy updating* scheme of the dependency graph, together with suitable data structures, keeps the updating efforts at bay, with several possibilities to track these dependencies.

7.4 One promising avenue of future research is to enhance the support of ML3-Rust for modeling various types of agents' decision processes, for example, involving Bayesian beliefs updating or other templates. The potential and limitations of ML3-Rust could also be tested in real simulation studies that address new research questions. These include the examples mentioned before, such as migration. Those could follow the preliminary analysis presented in Bijak et al. (2021) and Hinsch & Bijak (2022) or analyze the impact of different intervention strategies on recreational fishery (Haase et al. 2022).

7.5 Ongoing work on internal DSLs for continuous-time agent-based models is dedicated to the parallel execution of simulation runs. Comparing the performance of an optimistic synchronized parallel algorithm for tightly coupled continuous-time agent-based models (Andelfinger & Uhrmacher 2021) with a Timewarp implementation has revealed insights (Andelfinger et al. 2022). Those will be used to develop a hybrid algorithm combining synchronous and asynchronous mechanisms.

Acknowledgments

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant no. 725232 Bayesian Agent-based Population Studies. This article reflects the authors' views, and the Research Executive Agency of the European Commission is not responsible for any use that may be made of the information it contains. The work recieved funding from the Deutsche Forschungsgemeinschaft (DFG) grant number 258560741. We thank Tom Warnke for comments and contributing to discussions on the structure of the paper.

References

Abar, S., Theodoropoulos, G. K., Lemarinier, P. & O'Hare, G. M. P. (2017). Agent based modelling and simulation tools: A review of the state-of-art software. *Computer Science Review*, *24*, 13–33

Andelfinger, P., Piccione, A., Pellegrini, A. & Uhrmacher, A. M. (2022). Comparing speculative synchronization algorithms for continuous-time agent-based simulations. IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applicatiions (DS-RT 2022)

Andelfinger, P. & Uhrmacher, A. M. (2021). Optimistic parallel simulation of tightly coupled agents in continuous time. The 25th International Symposium on Distributed Simulation and Real Time Applocations (IEEE/ACM DS-RT 2021), Los Alamitos, CA, USA

Artho, C., Havelund, K., Kumar, R. & Yamagata, Y. (2015). Domain-specific languages with scala. International Conference on Formal Engineering Methods

Barringer, H. & Havelund, K. (2011). Internal versus external DSLs for trace analysis. International Conference on Runtime Verification

Bianchi, F. & Squazzoni, F. (2015). Agent-based models in sociology. *Wiley Interdisciplinary Reviews: Computational Statistics*, *7*(4), 284–306

Bijak, J., Higham, P. A., Hilton, J., Hinsch, M., Nurse, S., Prike, T., Reinhardt, O., Smith, P. W. F., Uhrmacher, A. M. & Warnke, T. (2021). *Towards Bayesian Model-Based Demography: Agency, Complexity and Uncertainty in Migration Studies*. Cham: Springer

Bonabeau, E. (2002). Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, *99*(3), 7280–7287

Buss, A. & Al Rowaei, A. (2010). A comparison of the accuracy of discrete event and discrete time. Proceedings of the 2010 Winter Simulation Conference, Piscataway, New Jersey

Caillou, P., Gaudou, B., Grignard, A., Truong, C. Q. & Taillandier, P. (2017). A simple-to-use BDI architecture for agent-based modeling and simulation. In W. Jager, R. Verbrugge, A. Flache, G. de Roo, L. Hoogduin & C. Hemelrijk (Eds.), *Advances in Social Simulation 2015*, (pp. 15–28). Berlin Heidelberg: Springer

Cao, Y., Li, H. & Petzold, L. (2004). Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. *The Journal of Chemical Physics*, *121*(9), 4059–4067

Collier, N. & North, M. (2013). Parallel agent-based simulation with Repast for high performance computing. *Simulation*, *89*(10), 1215–1235

Cordasco, G., Scarano, V. & Spagnuolo, C. (2018). Distributed MASON: A scalable distributed multi-agent simulation environment. *Simulation Modelling Practice and Theory*, *89*, 15–34

Danos, V. & Laneve, C. (2004). Formal molecular biology. *Theoretical Computer Science*, *325*(1), 69–110

Ewald, R. & Uhrmacher, A. M. (2014). SESSL: A domain-specific language for simulation experiments. *ACM Trans. Model. Comput. Simul.*, *24*(2), 25

Fowler, M. (2010). *Domain-Specific Languages*. London: Pearson Education

Funahashi, A., Matsuoka, Y., Jouraku, A., Morohashi, M., Kikuchi, N. & Kitano, H. (2008). Celldesigner 3.5: a versatile modeling tool for biochemical networks. *Proceedings of the IEEE*, *96*(8), 1254–1265

Gibson, M. A. & Bruck, J. (2000). Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry A*, *104*(9), 1876–1889

Gilbert, N. & Troitzsch, K. (2005). *Simulation for the Social Scientist*. New York, NY: McGraw-Hill Education

Gillespie, D. T. (1976). A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, *22*(4), 403–434

Gillespie, D. T. (1977). Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, *81*(25), 2340–2361

Gillespie, D. T. (2007). Stochastic simulation of chemical kinetics. *Annual Review of Physical Chemistry*, *58*, 35–55

Gowda, S., Ma, Y., Cheli, A., Gwóźdź, M., Shah, V. B., Edelman, A. & Rackauckas, C. (2022). High-performance symbolic-numerics via multiple dispatch. *ACM Communications in Computer Algebra*, *55*(3), 92–96

Haase, K., Weltersbach, M. S., Lewin, W. C., Strehlow, H. V., Reinhardt, O. & Uhrmacher, A. M. (2022). Site choice in recreational fisheries - Towards an agent-based approach. Winter Simulation Conference (WSC 2022)

Harris, L. A., Hogg, J. S., Tapia, J.-J., Sekar, J. A. P., Gupta, S., Korsunsky, I., Arora, A., Barua, D., Sheehan, R. P. & Faeder, J. R. (2016). Bionetgen 2.2: Advances in rule-based modeling. *Bioinformatics*, *32*(21), 3366–3368

Helms, T., Ewald, R., Rybacki, S. & Uhrmacher, A. M. (2015). Automatic runtime adaptation for component-based simulation algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, *26*(1), 1–24

Henzinger, T. A., Jobstmann, B. & Wolf, V. (2011). Formalisms for specifying markovian population models. *International Journal of Foundations of Computer Science*, *22*(4), 823–841

Hinsch, M. & Bijak, J. (2022). The effects of information on the formation of migration routes and the dynamics of migration. *Artificial Life*, *29*, 1–18

Keating, S. M., Waltemath, D., König, M., Zhang, F., Dräger, A., Chaouiya, C., Bergmann, F. T., Finney, A., Gillespie, C. S., Helikar, T. & others (2020). SBML Level 3: An extensible format for the exchange and reuse of biological models. *Molecular Systems Biology*, *16*(8), e9110

Kermack, W. O. & McKendrick, A. G. (1927). A contribution to the mathematical theory of epidemics. *Proceedings of the Royal Society of London: Series A*, *115*(772), 700–721

Keuschnigg, M., Lovsjö, N. & Hedström, P. (2018). Analytical sociology and computational social science. *Journal of Computational Social Science*, *1*(1), 3–14

Kleijnen, J. P. C. (2018). *Design and Analysis of Simulation Experiments*. Cham: Springer International Publishing

Köster, T., Giabbanelli, P. J. & Uhrmacher, A. (2020). Performance and soundness of simulation: A case study based on a cellular automaton for in-body spread of HIV. 2020 Winter Simulation Conference (WSC)

Köster, T. & Uhrmacher, A. M. (2018). Handling dynamic sets of reactions in stochastic simulation algorithms. Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation

Köster, T., Warnke, T. & Uhrmacher, A. M. (2022). Generating fast specialized simulators for stochastic reaction networks via partial evaluation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, *32*(2), 1–25

Kreikemeyer, J. N., Köster, T., Uhrmacher, A. M. & Warnke, T. (2021). Inferring dependency graphs for agent-based models using aspect-oriented programming. Winter Simulation Conference (WSC 2021)

Lafage, R. (2022). egobox, a Rust toolbox for efficient global optimization. *Journal of Open Source Software*, *7*(78), 4737

Lattner, C. & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. CGO, San Jose, CA, USA

Law, A. M. (2015). *Simulation Modeling and Analysis*. New York, NY: McGraw-Hill Education

Macal, C. M. (2016). Everything you need to know about agent-based modelling and simulation. *Journal of Simulation*, *10*(2), 144–156

Manson, S., An, L., Clarke, K. C., Heppenstall, A., Koch, J., Krzyzanowski, B., Morgan, F., O'Sullivan, D., Runck, B. C., Shook, E. & Tesfatsion, L. (2020). Methodological issues of spatial agent-based models. *Journal of Artificial Societies and Social Simulation*, *23*(1), 3

Matsakis, N. D. & Klock II, F. S. (2014). The Rust language. *ACM SIGAda Ada Letters*, *34*(3), 103–104

McCollum, J. M., Peterson, G. D., Cox, C. D., Simpson, M. L. & Samatova, N. F. (2006). The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behavior. *Computational Biology and Chemistry*, *30*(1), 39–49

Mendes, P., Hoops, S., Sahle, S., Gauges, R., Dada, J. & Kummer, U. (2009). Computational modeling of biochemical networks using COPASI. In I. V. Maly (Ed.), *Systems Biology*, (pp. 17–59). Berlin Heidelberg: Springer

Meyer, T., Helms, T., Warnke, T. & Uhrmacher, A. M. (2018). Runtime code generation for interpreted domain-specific modeling languages. 2018 Winter Simulation Conference (WSC)

Müller, J. P. & Pischel, M. (1993). The agent architecture INTERRAP: Concept and application. DFKI-Research Report RR-93-26. Saarbrücken

Niemann, J.-H., Winkelmann, S., Wolf, S. & Schütte, C. (2021). Agent-based modeling: Population limits and large timescales. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, *31*(3), 033140

North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M. & Sydelko, P. (2013). Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling*, *1*(1), 1–26

Özmen, O., Nutaro, J. J., Pullum, L. L. & Ramanathan, A. (2016). Analyzing the impact of modeling choices and assumptions in compartmental epidemiological models. *Simulation*, *92*(5), 459–472

Peters, F., Neuberger, D., Reinhardt, O. & Uhrmacher, A. (2022). A basic macroeconomic agent-based model for analyzing monetary regime shifts. arXiv preprint. arXiv:2205.00752

Railsback, S., Ayllón, D., Berger, U., Grimm, V., Lytinen, S., Sheppard, C. & Thiele, J. C. (2017). Improving execution speed of models implemented in NetLogo. *Journal of Artificial Societies and Social Simulation*, *20*(1), 3

Reinhardt, O. & Uhrmacher, A. M. (2017). An efficient simulation algorithm for continuous-time agent-based linked lives models. Proceedings of the 50th Annual Simulation Symposium, San Diego, CA, USA

Reinhardt, O., Uhrmacher, A. M., Hinsch, M. & Bijak, J. (2019). Developing agent-based migration models in pairs. Proceedings of the 2019 Winter Simulation Conference, Piscataway, New Jersey

Reinhardt, O., Warnke, T. & Uhrmacher, A. M. (2021). A language for agent-based discrete-event modeling and simulation of linked lives. *ACM Transactions on Modeling and Simulation*, *32*(1)

Schnoerr, D., Sanguinetti, G. & Grima, R. (2017). Approximation and inference methods for stochastic biochemical kinetics - A tutorial review. *Journal of Physics A: Mathematical and Theoretical*, *50*(9), 093001

Sheppard, C. J. R., Harris, A. & Gopal, A. R. (2016). Cost-effective siting of electric vehicle charging infrastructure with agent-based modeling. *IEEE Transactions on Transportation Electrification*, *2*(2), 174–189

Tisue, S. & Wilensky, U. (2004). NetLogo: Design and implementation of a multi-agent modeling environment. Available at: `https://ccl.northwestern.edu/papers/agent2004.pdf`

Uhrmacher, A. M., Tyschler, P. & Tyschler, D. (2000). Modeling and simulation of mobile agents. *Future Generation Computer Systems*, *17*(2), 107–118

Warnke, T. (2021). Domain-specific languages for modeling and simulation. University of Rostock. Available at: `http://eprints.mosi.informatik.uni-rostock.de/703/`

Warnke, T., Reinhardt, O., Klabunde, A., Willekens, F. & Uhrmacher, A. M. (2017). Modelling and simulating decision processes of linked lives: An approach based on concurrent processes and stochastic race. *Population Studies*, *71*(sup1), 69–83

Weimer, C., Miller, J. O., Hill, R. & Hodson, D. (2019). Agent scheduling in opinion dynamics: A taxonomy and comparison using generalized models. *Journal of Artificial Societies and Social Simulation*, *22*(4), 5

Wilensky, U. (1999). NetLogo. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. Available at: `http://ccl.northwestern.edu/netlogo/`

Willekens, F. (2017). Continuous-time microsimulation in longitudinal analysis. In A. Zaidi, A. Harding & P. Williamson (Eds.), *New Frontiers in Microsimulation Modelling*, (pp. 413–436). London: Routledge

Wilsdorf, P., Pierce, M. E., Hillston, J. & Uhrmacher, A. M. (2019). Round-based super-individuals - Balancing speed and accuracy. ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS 2019), New York, USA

Danksagung

Ich möchte diese Gelegenheit nutzen, um all jenen zu danken, die mich auf dem Weg zu dieser Dissertation unterstützt haben.

Besonders dankbar bin ich meiner Betreuerin, *Prof. Adelinde Uhrmacher*, deren unermüdliches Engagement und Unterstützung es mir ermöglicht hat, meinen eigenen Forschungsweg zu gehen. Sie hat mir nicht nur die Freiheit gegeben, meine Ideen zu verfolgen, sondern auch wertvolle Richtungen aufgezeigt, die meine Arbeit maßgeblich bereichert haben.

Ein großer Dank gilt auch meinen Kollegen, die mir stets mit Rat und Tat zur Seite standen. Besonders hervorheben möchte ich *Philipp Andelfinger*, von dem ich enorm viel lernen durfte, sowie *Pia Wilsdorf* und *Philipp Henning*, mit denen ich mir das Büro geteilt habe und die stets bereit waren, ihre Gedanken zu meiner Arbeit zu teilen. Mein Dank gilt auch an *Nadja Schlungbaum, Sigrun Hoffmann* und *Julia Prahl*, die mich stets souverän durch den Verwaltungsdschungel und technische Herausforderungen geführt haben.

Meinen Eltern gebührt mein tiefer Dank, denn ohne ihre fortwährende Unterstützung wären mein Studium und diese Arbeit nicht möglich gewesen.

Ich denke auch an meine Großeltern, von denen leider nicht alle die Fertigstellung meiner Arbeit miterleben durften. Ihr Glaube an mich und ihre stille Unterstützung haben mich tief geprägt. Danke, *Ursula, Walter, Gertrud* und *Lorenz*.

Meinen Freunden danke ich für die wunderbare Zeit, die durch gemeinsame Erlebnisse in Musik und Sport sowie unvergessliche Momente sowohl in als auch um Rostock herum bereichert wurde.

Ein ganz besonderer Dank gilt *Helena Drüeke*, die sich unermüdlich für mich und diese Arbeit eingesetzt hat und mein Leben wie kein anderer Mensch geprägt hat.

Ihnen allen gilt mein tiefster Dank. Sie haben dazu beigetragen, dass diese Arbeit nicht nur ein akademisches Projekt, sondern auch eine Zeit voller persönlicher Entwicklung und wertvoller Erinnerungen wurde.

Selbständigkeitserklärung

Ich habe die eingereichte Dissertation selbständig und ohne fremde Hilfe verfasst, andere als die von ihm/ihr angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht hat.

Till Köster, 30. Juni 2024