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.

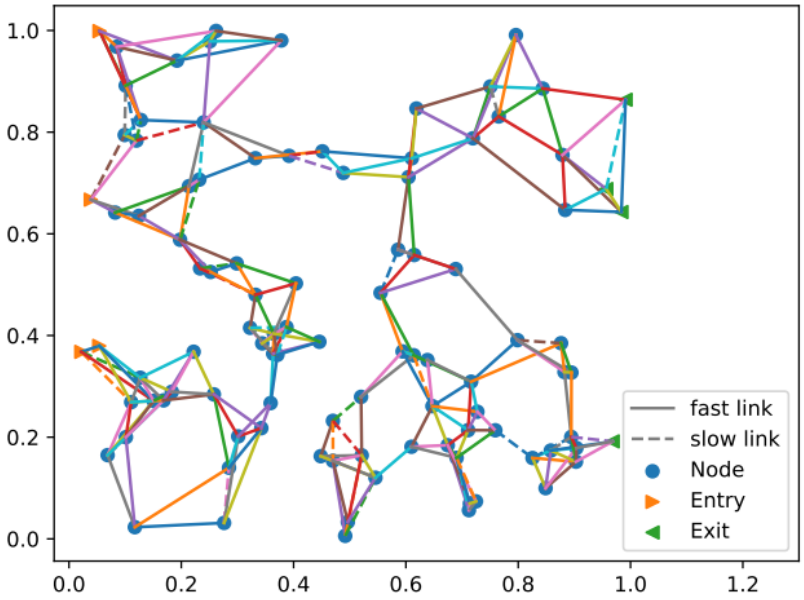


Figure 4: A randomly generated spatial network. The agents travel from Entry to Exit across fast and slow links.

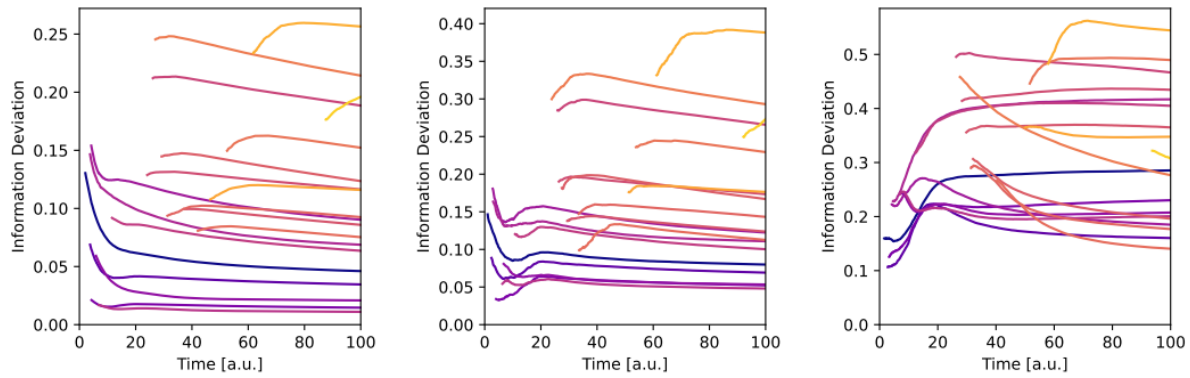
Agents, communication, and information

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

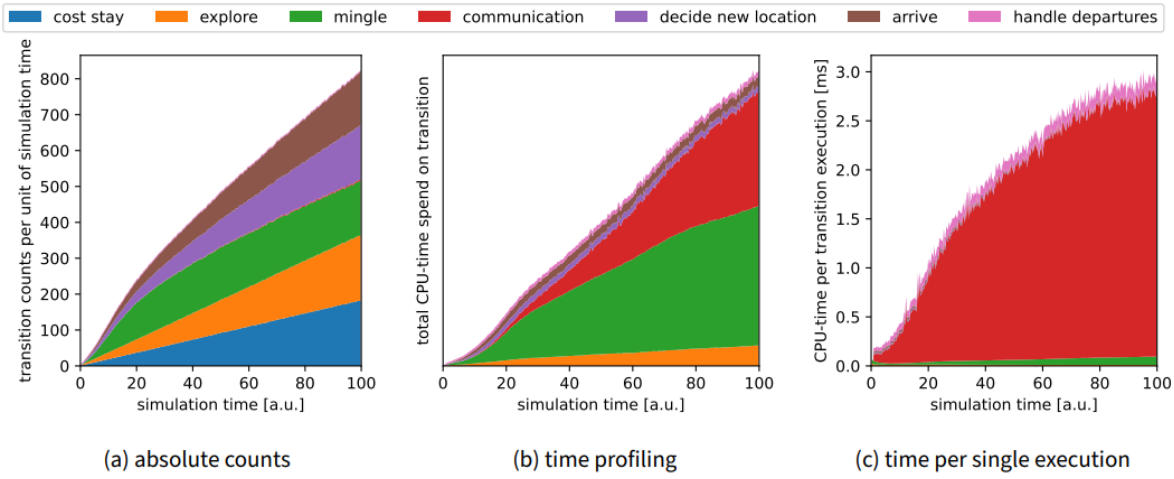
Figure 5: The deviation of the average knowledge about different locations across agents through time for different errors of transmission. The results are averaged across 500 replications. Simulation experiments of this scale were not possible with the previous version of ML3.

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.



(a) absolute counts (b) time profiling (c) time per single execution

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:

$$S \xrightarrow{\sum_{I \in \text{Neuman}} 5 \cdot 10^{-2} + 2 \cdot 10^{-6}} I$$

$$I \xrightarrow{10^{-2}} R$$

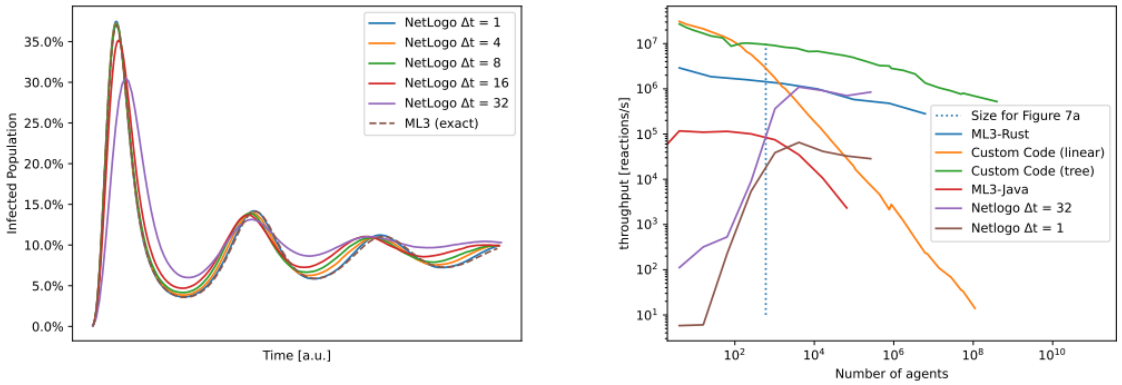
$$R \xrightarrow{1.5 \cdot 10^{-3}} S$$

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