AN EFFICIENT SIMULATION ALGORITHM FOR CONTINUOUS-TIME AGENT-BASED LINKED LIVES MODELS

Oliver Reinhardt
Adelinde M. Uhrmacher

Institute of Computer Science
University of Rostock
Albert-Einstein-Str. 22
D-18059 Rostock, GERMANY
{firstname.lastname}@uni-rostock.de

ABSTRACT

The efficient continuous-time simulation of linked lives in demography implies specific challenges. The resulting agent-based models constitute time-inhomogeneous Markov chains which require stochastic simulation algorithms. Each agent is characterized by diverse attributes, including a specific position in a dynamically evolving social network which influences the agent's behavior. This hampers the application of population-based approaches in implementing the stochastic simulation algorithm. However, as events are locally constrained by the social network, many events will happen independently of each other. We develop a stochastic simulation algorithm that maintains a dependency structure to realize lazy re-calculation of events. In case study on a Susceptible-Infected-Recovered-Model with social network and age-dependent susceptibility we evaluate the performance of the algorithm in comparison to an earlier version. The evaluation shows the improved scalability and a significant speedup of up to 150 times that can be achieved by taking dependencies into account when executing linked, continuous-time agent-based models.

Keywords: agent-based simulation, stochastic simulation algorithms, time-inhomogeneous Markov Chains.

1 INTRODUCTION

Over the last two decades agent-based models have become an established method in modeling and simulation. With more convenient ways to specify agent-based models, e.g., (Warnke et al. 2015, Warnke et al. 2016, Birdsey et al. 2016), those models tend to become more and more complex (North et al. 2013), and with the size of the models their efficient execution becomes more challenging. Most agent-based models are still executed in a step-wise manner, however, the number of multi-agent models being executed by discrete event approaches is increasing. Scheduling events in continuous time allows capturing the temporal behavior of multi-agent systems more realistically (Willekens 2009). Therefore, widely used agent-based modeling and simulation systems have started to provide rudimentary support for this kind of continuous-time agent-based models (Warnke et al. 2016), e.g., both Repast Simphony as well as NetLogo allow events to be manually scheduled and retracted as part of the model (Sweda and Klabjan 2011, Sheppard and Railsback 2015). Of particular interest are continuous time models where the sojourn times are sampled from exponential distributions. Applications range from chemical models (Gillespie 1977), epidemical models (Allen and Lahodny 2012), to decision processes of individuals to migrate (Klabunde et al. 2015). For the chemical realm the Doob-Gillespie algorithms (Doob 1945, Gillespie 1977) have become an established

SpringSim-ANSS 2017, April 23-26, Virginia Beach, VA, USA
©2017 Society for Modeling & Simulation International (SCS)

means to simulate those models and have also been applied for executing other continuous-time agent-based models (Vestergaard and Génois 2015). However, whereas in bio-chemical applications agents can often be aggregated to populations or even concentrations in the continuous limit (Geisweiller et al. 2008), in demographic models of linked lives this is not the case.

In demographic models of linked lives, each individual tends to be unique by its birth and the social network the individual is part of, e.g., family ties. Its behavior is conditional on its network and its own properties. Its age plays a central role in its possibilities and decisions, e.g., for marriage and child birth. This leads to the exponential distribution of the sojourn times of the continuous-time agent-based models being time-dependent (Hoem et al. 1976). All of this implies, that although the basic Doob-Gillespie algorithm is applicable it might not be very efficient, and new solutions are required.

In the following we will present a new simulation algorithm for executing this type of continuous-time agent-based models more efficiently. Thereby, we will base our efforts on the highly expressive Modeling Language for Linked Lives (ML3) (Warnke et al. 2015), which is aimed at succinctly describing the diverse decision processes of individuals in continuous time.

After briefly presenting ML3, we sketch the basic Doob-Gillespie simulation algorithm of ML3, and discuss the implications of applying this algorithm. To develop the new algorithm we will exploit locality and dependencies for designing a more efficient simulator which is a reminiscent of a particular variant of the Doob-Gillespie algorithms, i.e., the Next Reaction Method. We will present the challenges induced also due to the expressiveness of the domain specific language and the realized means to address those in our new simulator which will be put to test in a small evaluation study. The evaluation is based on three topological variants of the SIR model. The performance is compared to the basic implementation. The paper concludes with a summary and open questions.

2 THE ML3 MODELING LANGUAGE

The Modeling Language for Linked Lives (ML3) is a domain specific modeling language that is specifically designed to describe continuous-time agent-based models in demography, where agents interact in a social network. An ML3 model consists of three kinds of components: agents, links between agents, and rules. Every model entity that acts independently is represented as an *agent*. This does not only include persons, but also higher-level actors such as families and households. A higher-level actor all other agents interact with could be a global continuous space all other agents roam in (Helleboogh et al. 2007), thus mimicking a NetLogo world (Sheppard and Railsback 2015). However, the focus of ML3 is clearly on networks constituting the environment of agents. This approach can also be beneficial and very expressive to model spatial interactions and behaviors of agents, as illustrated by a recent work on collective adaptive systems that exploits a set of named, discrete, and related locations as the underlying model of space (Feng, Hillston, and Galpin 2016). As different kinds of entities get modeled as agents, every agent has a type that determines its properties and behavior. The first part of every ML3 model contains the definitions of the agent types. Each definition consist of the agent type's name and a set of typed attributes. The following code snippet defines an agent type `Person` with three attributes. The first attribute, `status`, can take one of three values. The person is either a child, an adult, or retired. The other two attributes' values are real numbers.

```
1 Person(
2   status : {"child", "adult", "retired"},
3   savings : real,
4   income : real
5 );
```

In addition to the attributes defined by its type every agent has an age. It behaves like an attribute, but its value changes automatically when time passes. Also, every agent is either alive or dead. Agents are alive when they are created and can die as part of the model's behavior.

As ML3 is primarily designed for models in which the acting entities are part of networks that influences their behavior, these networks are an explicit part of a ML3 model. Every relationship between agents is modeled by a *link*. For example, the members of a household are connected to the household by a link. Parents are connected to their children by a link. Children are also connected to their parents by a link. A link definition includes the types of the linked agents, the *role names* of the involved agents, and a *link cardinality* that states the number of partners a agent can be linked to. The following snippet shows the definition of a link:

```
1 members:Person[1-] <-> [1]Household:household;
```

This link definition defines the link between a person and its household. It states that each person is connected to exactly one household. The person refers to this household by the role name `household`. In the opposite direction every household is connected to one or more persons, that are referred to as `members`. Links are always bidirectional and when a link is changed in one direction, the same change is automatically applied for the other direction. When a child moves away from its parents, its household will be changed to another household. In this case it will automatically be removed as a member of the parent's household.

How attributes and the network of an agent change over time, its behavior, is described by *rules*. Every rule applies to all agents of a specific agent type that are currently alive. Rule definitions consist of three parts: the *guard condition*, the *waiting time expression* and the *effect*. The guard condition specifies *who* is affected by the rule, i.e., which agents of this agent type are affected. The waiting time expression specifies *when* the rule is executed. Finally, the effect describes *what* happens, when the rule is executed. The following rule describes how a person moves out of its parents' household:

```
1 Person
2 | ego.status = "adult", // guard condition
3   ego.household = ego.mother.household
4 @ moveOutRate // waiting time expression
5 -> ego.household := new Household(); // effect
```

This rule applies to agents of the agent type `Person`. Every agent of this type has a *rule instance* of this rule that applies to it. The agent is referred to by the keyword **`ego`**, similar to the keyword **`this`** in many object oriented programming languages. The guard condition specifies that it is only active for agents who are currently adults and live in the same household as their mother.

The waiting time expression describes the waiting time until the rule is executed. The waiting times between the executions of a stochastic rule like the above are the interarrival times of an inhomogeneous Poisson process. The value of the waiting time expression gives the arrival rate of the Poisson process. In this case the rate is given by the model parameter `moveOutRate`.

The third part of the rule is its effect. Rule effects are specified in an imperative style, as a list of statements that is executed sequentially. In this case the agents status is changed to infected. Beyond the change of attributes and links possible rule effects include the creation and death of agents. We have already seen the creation of an agent in the above example, where a new household is created. The death of an agent can be triggered by calling the predefined procedure `die`.

The semantics of ML3 are based on time-inhomogeneous continuous-time Markov chains (CTMC). A state of the CTMC is given by a set of agents with their attributes, their links, and the times of their creation. Each

```
1  Person(status : {"susceptible", "infectious", "recovered"});
2
3  network:Person[0-]<->[0-]Person:network;
4
5  Person
6  | ego.status = "susceptible"
7  @ a * ego.network.filter(alter.status = "infectious").size() *
       infectionScaling[ego.age]
8  -> ego.status := "infectious";
9
10 | ego.status = "infectious"
11 @ b
12 -> ego.status := "recovered";
```

Figure 1: A SIR model with age-dependent susceptibility in ML3.

rule of the model describes a set of state transitions of the CTMC. In principle, each agent of the type for which the rule is defined may contribute a transition, as the new state is different, depending on the agent to which the rule was applied. While multiple agents might theoretically share the same attribute values, link partners and time of creation, in praxis this will only happen rarely. Transition rates are given by the rate expressions. As these can depend on time the CTMC is time-inhomogeneous. While multiple agents might theoretically share the same attribute values, link partners and time of creation, in praxis this will only happen rarely.

3 A SIR MODEL IN ML3

To illustrate the language and later on the simulation algorithms we will now introduce an implementation of a simple SIR model in ML3. SIR models are a kind of population-based epidemiological models, where individuals change between the states *susceptible*, *infectious* and *recovered* (Allen and Lahodny 2012). Susceptible individuals can get infected, i.e., transition from the susceptible to the infectious state, with a rate depending on the infection rate constant a of the disease and the number of infectious individuals in the population. The general idea is that a person has a certain chance to fall ill whenever they meet a infectious person. Infectious individuals may recover, i.e., transition from the infectious to the recovered state, with a recovery rate constant b depending on the disease. We expand the standard SIR model in two aspects. Firstly, we assume that persons do not make contact with every other person in the population, but only with people from their social network, i.e., their colleagues, friends, family, and other persons they contact regularly. Therefore they can only get infected by people from their social network. This aspect of our model is inspired by the BioWar model (Carley et al. 2006), but much simpler. Secondly, we let the infection rate vary with the individual's age. Age-dependent susceptibility is used to consider the development of the immune system in a SIR model (Korobeinikov and Melnik 2013).

Figure 1 shows the complete ML3-implementation of that model. The acting individuals are represented as agents. In this case we have one type of agents, `Person` (line 1). Each person has a status, that takes one of the three values susceptible, infectious and recovered. The social network is represented as a link between agents (line 3). Every person is linked to an arbitrary number of other persons, their `network`. Note that the role name is the same for both directions of the link, making it an undirected link, as opposed to the previous example of the link between persons and a household.

The rest of the model consists of two rules for the agent type `Person`. The infection rule (line 6-8) shall apply to all susceptible persons (line 6). It changes the person's status to infectious (line 8). Its rate is determined by the infection rate constant a, the number of infectious network neighbors, and an age-dependent scaling factor. The number of infectious network neighbors is determined by taking the set of the agent's network neighbors (`ego.network`), filtering for the infectious ones (`.filter(alter.status = "infectious")`), and taking the size of the remaining set as a result (`.size()`). The age-dependent scaling factor is implemented as a map, a data structure ML3 uses for time series data. Similarly the recovery rule (line 10-12) implements the recovery of an infectious person.

4 THE STOCHASTIC SIMULATION ALGORITHM

Simulation algorithms for stochastic models with CTMC semantics are a well researched area. The Stochastic Simulation Algorithm (SSA) was originally introduced by Doob (Doob 1945) and popularized by Gillespie (Gillespie 1977). Gillespie designed the algorithm for systems of chemical reactions, but it applies for all systems with CTMC semantics. Given a state s of the Markov Chain at certain time t, the algorithm calculates the next state s' of the system and the time $t' = t + \Delta t$ it enters this next state. This way a trajectory of the Markov Process is sampled.

Essentially the SSA achieves this by sampling the waiting time distribution of every state transition of the CTMC and executing the transition with the shortest waiting time. The waiting time T is exponentially distributed:

$$P(T \leq \Delta t) = 1 - \exp(-\lambda(s)\Delta t) \tag{1}$$

In the usual case of a homogeneous CTMC the exponential distribution's rate parameter $\lambda(s)$ only depends on the current state s. This distribution can be efficiently sampled by applying the inversion method, i.e., sampling r from the uniform distribution on the unit interval and applying the distribution function's inverse:

$$\Delta t = \frac{1}{\lambda(s)} \cdot \ln \frac{1}{r} \tag{2}$$

In ML3's case of transition rates that can additionally depend on simulation time, the SSA can be applied in largely the same way (Jansen 1995). Transitions rates $\lambda(s,\tau)$ are now no longer constant between state changes, leading to a similar but more complex waiting time distribution:

$$P(T \leq \Delta t) = 1 - \exp(-\int_t^{t+\Delta t} \lambda(s,\tau)\, d\tau) \tag{3}$$

This distribution is a generalized form of the exponential distribution. When $\lambda(s,\tau)$ is constant as a function of τ the value of the integral is $\lambda(s)\Delta t$, as in equation (1). Applying the inversion method leads to the following equation:

$$\int_t^{t+\Delta t} \lambda(s,\tau)\, d\tau = \ln \frac{1}{r} \tag{4}$$

For general functions of time, an analytical solution of this equation is unknown. Even for time-dependent rate functions that are commonly used in demography like the Gompertz-Makeham law of mortality a closed

form solution gets very complex (Jodrá 2009). For reasons of efficiency numerical solutions are also unfeasible, as a large number of waiting times has to be calculated. In the case of rate expressions that are piecewise constant in time, the integral is a piecewise linear function for which an analytical solution of the equation can be found. More complex rate functions can be approximated by piecewise constant functions. As rates for demographic events like mortality and fertility are often estimated from data that is gathered yearly and for age cohorts and is therefore inherently piecewise constant, we limit ML3 to these. The algorithms we describe in this paper are not affected by this constraint. They apply in the same way when the waiting time distribution is sampled in different ways that allows for more complex functions of time.

```
1  t = 0
2  while t < t_end do
3   delta_t = inf
4   activated_r = null
5   activated_a = null
6   for each a in alive_agents(s) do
7     for each r in rules(type(a)) do
8       if guard_satisfied(a,r,s) do
9         delta_t_r_a = waiting_time(a,r,s,t)
10        if delta_t_r_a < delta_t do
11          delta_t = delta_t_r_a
12          activated_r = r
13          activated_a = a
14  t = t + delta_t
15  execute(activated_a,activated_r,s,t)
16 end
```

Figure 2: The stochastic simulation algorithm applied to ML3.

Figure 2 shows how the SSA can be directly applied to ML3. Every active ML3 rule instance, i.e., every pair of an agent and a rule that applies to this agent, contributes a transition. Therefore for each agent the simulator determines the set of all rules that are defined for this agent type. For each of these rules the guard condition is evaluated. When the guard evaluates to **`true`**, the waiting time according to the waiting time expression is determined. If it is smaller than the waiting time of the currently chosen rule instance, this instance is chosen instead.

For agent-based models with heterogeneous populations of agents this algorithm is highly inefficient. The algorithm was originally developed for systems with a limited number of possible transitions. In each step a waiting time for each transition of the Markov chain has to be calculated to execute a single event. Heterogeneous populations of agents lead to a very large number of transitions, as the application of the same change to different agents results in different states. The number of transitions, and therefore the time required to execute a single event, grows linearly with the number of agents. In consequence this algorithm is not feasible for large numbers of agents.

5 EXPLOITING LOCALITY

As shown in the previous section the standard SSA is inefficient for models with populations of highly heterogeneous agents, as the number of events that have to be scheduled in every step is very high. We will now continue to show that the amount of events that have to be rescheduled in every step can be reduced significantly, when the effects of most events are restrained to local changes.

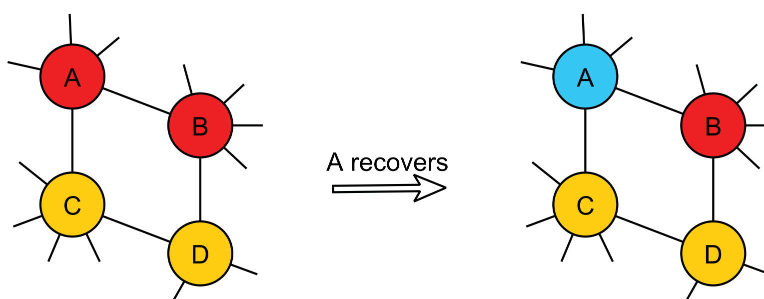

Figure 3: An extract of a possible SIR model state (yellow – susceptible, red – infectious, blue – recovered).

Consider the SIR model (Section 3) with a population of 1,000 agents and the state depicted on the left side of Figure 3. Following the naive SSA we have to consider 2,000 rule instances to find the next event, as we have to consider infection and recovery of all agents to find the one with the minimal waiting time. Let us assume that is the recovery of A. After the execution of that event we have a new state, as depicted on the right side of Figure 3. At this point we have finished the calculation of one event and start with the next one. For that we have to evaluate guard conditions for all 2,000 rule instances and draw a waiting time for all the active instances once again. However, the recovery of A only effects a small part of the state. As all rules of the model only take direct network neighbors of **ego** into consideration, only rule instances of the five direct neighbors of A are affected. For most rule instances the evaluation of the guard will give the same result and the execution time will be drawn from the same distribution as in the previous step. So instead of recalculating them we could reuse these previously calculated times and only reschedule the execution of the few affected rule instances to gain efficiency.

This idea of reusing the execution time generated by the SSA is not new, but goes back to the Next Reaction Method (NRM) introduced by Gibson and Bruck (Gibson and Bruck 2000) for chemical reaction networks. They introduced a dependency graph with all possible reactions as its nodes and a directed edge from a reaction r_i to another reaction r_j when the execution of r_i leads to a change of the reaction rate of r_j. Initially the algorithm generates execution times for all reactions like the original SSA. These are stored in an event queue, a priority queue that uses the execution times as priorities. The reaction with the smallest execution time is then retrieved from the queue and executed. Afterwards the dependency graph can be used to determine all reactions that have to be rescheduled. For these reactions a new execution time is determined and their position in the event queue is updated accordingly. It has been shown that the NRM is significantly more efficient than the original SSA when the firing of one reaction does not affect many other reactions, i.e., when the effects of events are mostly local (Cao et al. 2004). However, this approach is not directly applicable to agent based models with evolving social networks between the agents. In the case of chemical reaction network the dependency graph is relatively small, easy to generate, and static. The size of the graph is determined by the number of possible transitions in each state, as each transition contributes one node to the graph. With heterogeneous populations of agents the number of transitions is typically very large, as we have already argued. We need to distinguish between the different transitions that different instances of a ML3-rule contribute. In a set of chemical reactions it is easy to determine the reactions that are dependent on a given one. The execution of $A+B \rightarrow C$ changes the number of A, B, and C in the system, and nothing else. So only reactions with A, B or C as reactants are affected. In ML3 transition rates can be much more complex, e.g., rates depending on attributes of agents that are reached via multiple links. Also, rule effects may be intricate, e.g., changing attributes and links of multiple agents, not just the one that the rule was applied to. Therefore the task creating the dependency graph by analyzing the rules on a syntactical level is less straight forward. In addition, the network structures evolve during the simulation, as ML3 allows for the

```
1  schedule(a, r, s, t, q)
2    (b, p1) = evaluate_guard(a, r, s)
3    if b do
4       (delta, p2) = waiting_time(a, r, s, t)
5       q = requeue(a, r, t+delta)
6    log_dependencies(a, r, p1, p2)
```

Figure 4: Pseudocode for scheduling an event: `schedule(a, r, s, t, q)`. For clarity we show the logging of dependencies as a separate step.

```
1  t = 0; q = null
2
3  for each a in alive_agents(s) do
4    for each r in rules(type(a)) do schedule(a, r, s, t, q)
5  end
6
7  while t < t_end do
8    (a, r, t) = top(q); (s', p) = execute(a, r, s, t); s = s'
9    for each (agent, attr, link) in p do
10     for each (a, r) in dependent_events(agent, attr, link) do
11        schedule(a, r, s, t, q)
12   for each a in created(p) do
13     for each r in rules(type(a)) do schedule(a, r, s, t, q)
14   for each a in died(p) do
15     for each r in rules(type(a)) do retract (a, r, q)
16 end
```

Figure 5: The rule instance with the smallest time stamp is selected and executed. The part of the state that is changed thereby is recorded in `p`. All rule instances that are dependent on `p` need to be rescheduled.

creation and removal of agents and the change of links during the simulations. So with ML3 a dependency graph would not be static; it has to be updated after every event. We introduce a different way to manage the dependencies between rule instances. A rule instance has to be rescheduled when the specific part of the state that affects the value of the guard and rate expression of this rule instance is changed. When we schedule a rule instance we will determine all parts of the state that affect this rule instance. When a rule instance is executed we determine which parts of the state change during the execution. Then we reschedule all rule instances that depend on the part of the state that is changed.

The state of an ML3 model is a set of agents, with specific values for their attributes, links and aliveness. Therefore the values of guard and rate expressions can only change, when attributes, links and aliveness of agents change that are evaluated as part of the guard and rate expression being evaluated. Their change might not necessarily change the value of the guard or rate expression. However, the value of the guard or rate expression will definitely remain the same if the part of the state that is accessed by evaluating the guard and rate expression has not changed since last accessing it.

We use this idea to create a data structure that captures the dependencies between rule instances and attributes and links. The model state is, as in the naive SSA approach, stored as a set of agents with the

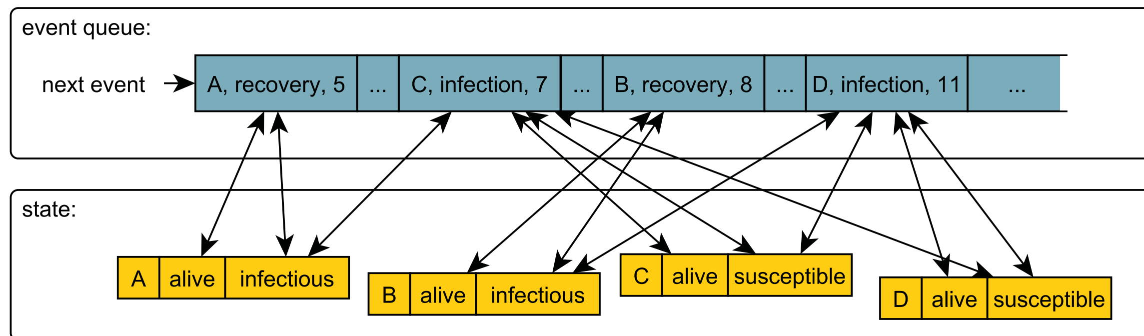

Figure 6: The dependency data structure for the example in Figure 3. The top shows the event queue, the bottom shows the state as a set of agents. Unscheduled rule instances are not shown.

associated attributes and links, times of creation and death. Additionally we store pointers to all depending rule instances together with attribute and link values and the information about the aliveness of the agent. To enable us to remove dependencies during rescheduling we also always set a backward pointer from the dependent rule instance to the attribute, link, or aliveness. All rule instances are scheduled in an event queue, similar to the NRM. To schedule a rule instance we do the following (Figure 4): Firstly, the guard condition is evaluated. During the evaluation of the guard, whenever we evaluate an attribute, link or the aliveness of an agent, we add a pointer to the rule instance. When the guard evaluates to false, we do not insert the rule instance into the event queue, or remove the rule instance from the queue, if it is currently scheduled. The rule instance itself is then kept in memory, so the pointers to it remain valid. If the guard evaluates to true, we draw from the waiting time distribution. Again we add this instance as a dependency to all attributes and links we have evaluated while calculating the waiting time. Then we add the rule instance together with the now determined waiting time to the event queue.

We can now determine the next event by taking the top element of the event queue. During the execution of the event we keep track of the attributes and links of agents that are changed. In addition we record the creation and death of agents. Now we retract all instances belonging to agents that died, as they are no longer active. Then we look up the rule instances that depend on the changed part of the state and reschedule them. To do that we remove all pointers in the data structure to the rule instance we want to reschedule, using the backwards pointers, and then schedule it again. Figure 5 shows this in pseudocode.

Figure 6 shows an example of this. Once again, consider the situation of Figure 3. Currently the state is as depicted on the left. When scheduling the rule instance for the infection of C, we needed to evaluate the attribute `status` of C (for the guard) and all it's neighbors (for the rate), including A and D. Therefore the status attribute of these agents is linked to that rule instance in the queue. We have done the same for all other rule instances. Now we can execute the next event, the recovery of A. When we do that, the attribute `status` of A changes. To determine which events need to be rescheduled we look up the events that are dependent on A's status: The infection and recovery of A, and the infection of C. We do not need to reschedule any other events.

As we no longer need to reschedule all rule instances after each event, the number of recalculated waiting times after each events no longer grows linear with the number of agents. When we assume that the effects of events are strongly local, as is the case in our SIR model, and the size of network neighborhoods does not increase with the number of agents, the number of rescheduled rule instances no longer depends on the population size. Even when this assumption does not hold, the number of rescheduled events is still considerably smaller, as long es not all events have global effects. In addition to that we have to consider

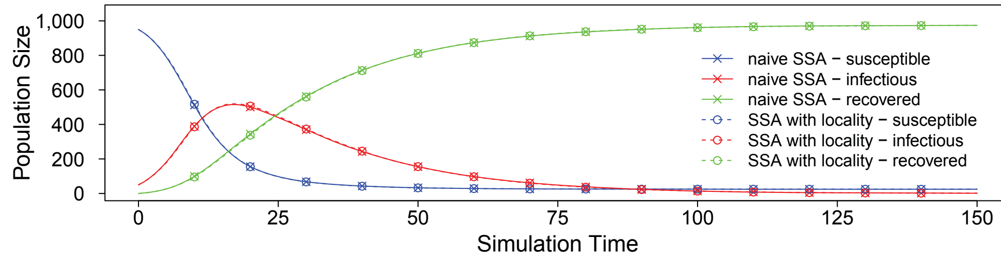

Figure 7: Average number of susceptible, infectious and recovered agents.

the costs of managing the dependency data structure and the event queue. The former adds an overhead to the evaluation of guard and waiting time expression, and after the execution of the event. This overhead is independent of the population size. The cost of the latter depends of the implementation of the event queue and the amount of locality in the model (Jeschke and Ewald 2008). Altogether we expect our algorithm to be more efficient than the naive SSA for large populations of agents.

6 EVALUATION

To evaluate our approach, we executed some experiments using the SIR model from Section 3. We implemented both the original SSA and our improved algorithm for ML3 in Java. In a first experiment we show that both algorithms yield the same results. The second experiment shows that the exploitation of locality indeed improves scalability in terms of the number of agents. In the third experiment we show that this advantage diminishes when locality gets weaker due to additional network links, but an advantage remains as long as the average number of network neighbors remains reasonable low.

The ML3 code in Section 3 contains only the model structure and behavior, but not the parameterization and the initial state. For our first experiment we created a population of 1000 agents with ages uniformly distributed between 0 and 100, of which 5% are initially infected. To establish a network between the agents we choose five random neighbors for each agent. This way each agent ends up with five neighbors, when no other agent chooses this one as a neighbor, or more, when other agents choose this one as a neighbor. We set the parameters a and b to 0.03 and 0.05 respectively and the age-dependent scaling of the infection rate to 0.5 for persons younger than 20 years, 1 between the ages 20 and 60, and 2 for everyone older than 60 years. We end a simulation run when no infected agents are left. With this configuration we executed 100 replications each with both with the simulation algorithms. Figure 7 shows the results for both configurations. We see the wave of infection that is typical for SIR models.

To evaluate the performance of our algorithm and compare it to the naive approach we varied the population size and the number of links and measured the time needed to execute a single replication. Figure 8 shows the results. The variation of the population size shows that our approach has a significant advantage in scalability versus the naive approach (left side). While it is less than 3 times as fast with a population of 100 agents and five links per agent, it's already 150 times faster with 10,000 agents. However, this advantage depends strongly on the network density (right side). With increased number of links, events are less locally constrained, so that the gains from exploiting locality diminish. At some point they get smaller than the overhead due to the need to maintain the event queue. However, in a real-life demographic model we expect a large population of at least a few thousand agents, while each agent has only a few link partners.

7 CONCLUSION

We present a variation of the Stochastic Simulation Algorithm for the modeling language ML3, a modeling language for agent-based demographic models with populations of agents interacting in a network. While

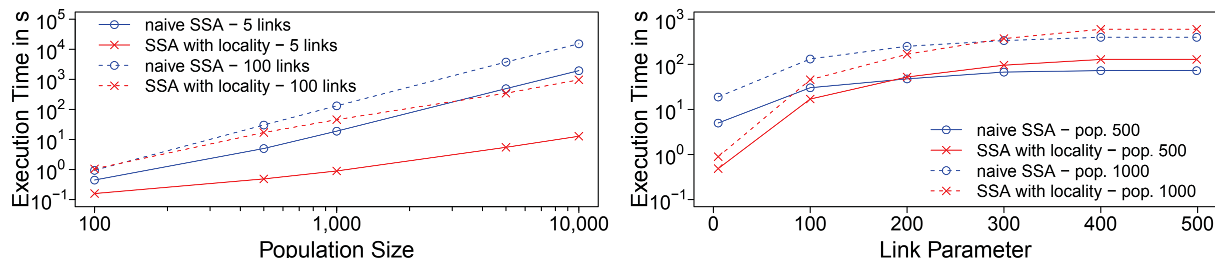

Figure 8: Comparison of the average execution time of one replication for the two algorithms as a function of population size (left) and number of links (right).

we focus on ML3, our approach is applicable for similar models, i.e, where the majority of agents need to be treated individually, implemented in other formalisms. We showed that the SSA in its usual form is highly inefficient for this kind of models, as due to the heterogeneity of the agent population the number of transitions that need to be scheduled after every event is very large. However, as an agent does not interact with all agents but only with a fraction of the agent population, effects of events tend to be local. We exploit this locality property by introducing a data structure to keep track of dependencies between transitions, so we only need to reschedule them when the waiting time actually changes. We demonstrate the feasibility of our algorithm by applying it to a SIR model of infectious disease in a small case study. Future work may include other methods to sample the waiting time distribution to allow for more general time-dependent rate functions and the application of approximation to further speed up the simulation.

ACKNOWLEDGMENTS

This research is partly supported by the German Research Foundation (DFG) via research grant UH-66/15-1.

REFERENCES

Allen, L. J. S., and G. E. Lahodny. 2012. “Extinction Thresholds in Deterministic and Stochastic Epidemic Models”. *Journal of Biological Dynamics* vol. 6 (2), pp. 590–611.

Birdsey, L., C. Szabo, and K. Falkner. 2016. “CASL: A Declarative Domain Specific Language For Modeling Complex Adaptive Systems”. In *Proceedings of the 2016 Winter Simulation Conference*. IEEE Press.

Cao, Y., H. Li, and L. Petzold. 2004. “Efficient Formulation of the Stochastic Simulation Algorithm for Chemically Reacting Systems”. *The Journal of Chemical Physics* vol. 121 (9), pp. 4059–4067.

Carley, K., D. Fridsma, E. Casman, A. Yahja, N. Altman, Li-Chiou Chen, B. Kaminsky, and D. Nave. 2006. “BioWar: Scalable Agent-Based Model of Bioattacks”. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* vol. 36 (2), pp. 252–265.

Doob, J. L. 1945. “Markoff Chains–Denumerable Case”. *Transactions of the American Mathematical Society* vol. 58 (3), pp. 455.

Feng, C., J. Hillston, and V. Galpin. 2016. “Automatic Moment-Closure Approximation of Spatially Distributed Collective Adaptive Systems”. *ACM Trans. Model. Comput. Simul.* vol. 26 (4), pp. 26:1–26:22.

Geisweiller, N., J. Hillston, and M. Stenico. 2008. “Relating continuous and discrete PEPA models of signalling pathways”. *Theoretical Computer Science* vol. 404 (1-2), pp. 97–111.

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* vol. 104 (9), pp. 1876–1889.

Gillespie, D. T. 1977. “Exact Stochastic Simulation of Coupled Chemical Reactions”. *The Journal of Physical Chemistry* vol. 81 (25), pp. 2340–2361.

Helleboogh, A., G. Vizzari, A. M. Uhrmacher, and F. Michel. 2007. “Modeling dynamic environments in multi-agent simulation”. *Autonomous Agents and Multi-Agent Systems* vol. 14 (1), pp. 87–116.

Hoem, J. M., N. Keiding, H. Kulokari, B. Natvig, O. Barndorff-Nielsen, and J. Hilden. 1976. “The Statistical Theory of Demographic Rates: A Review of Current Developments [with Discussion and Reply]”. *Scandinavian Journal of Statistics* vol. 3 (4), pp. 169–185.

Jansen, A. P. J. 1995. “Monte Carlo Simulations of Chemical Reactions on a Surface with Time-Dependent Reaction-Rate Constants”. *Computer Physics Communications* vol. 86 (1), pp. 1–12.

Jeschke, M., and R. Ewald. 2008. “Large-Scale Design Space Exploration of SSA”. In *Computational Methods in Systems Biology*, edited by M. Heiner and A. M. Uhrmacher, Volume 5307, pp. 211–230. Springer Berlin Heidelberg.

Jodrá, P. 2009. “A Closed-Form Expression for the Quantile Function of the Gompertz–Makeham Distribution”. *Mathematics and Computers in Simulation* vol. 79 (10), pp. 3069–3075.

Klabunde, A., F. Willekens, S. Zinn, and M. Leuchter. 2015. *An Agent-Based Decision Model of Migration, Embedded in the Life Course - Model Description in ODD+D Format*. MPIDR Working Paper WP-2015-002. Max Planck Institute for Demographic Research.

Korobeinikov, A., and A. V. Melnik. 2013. “Lyapunov Functions and Global Stability for SIR and SEIR Models with Age-Dependent Susceptibility”. *Mathematical Biosciences and Engineering* vol. 10 (2), pp. 369–378.

North, M. J., N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko. 2013. “Complex Adaptive Systems Modeling with Repast Simphony”. *Complex Adaptive Systems Modeling* vol. 1 (1), pp. 3.

Sheppard, C. J. R. and Railsback, S. 2015. “Time Extension for NetLogo (Version 1.2)”. https://github.com/colinsheppard/time. Accessed July 2016.

Sweda, T., and D. Klabjan. 2011. “An Agent-Based Decision Support System for Electric Vehicle Charging Infrastructure Deployment”. In *2011 IEEE Vehicle Power and Propulsion Conference*, pp. 1–5. IEEE.

Vestergaard, C. L., and M. Génois. 2015. “Temporal Gillespie Algorithm: Fast Simulation of Contagion Processes on Time-Varying Networks”. *PLOS Computational Biology* vol. 11 (10), pp. e1004579.

Warnke, T., O. Reinhardt, and A. M. Uhrmacher. 2016. “Population-Based CTMCs and Agent-Based Models”. In *Proceedings of the 2016 Winter Simulation Conference*. IEEE Press.

Warnke, T., A. Steiniger, A. M. Uhrmacher, A. Klabunde, and F. Willekens. 2015. “ML3: A Language for Compact Modeling of Linked Lives in Computational Demography”. In *Proceedings of the 2015 Winter Simulation Conference*, pp. 2764–2775. IEEE Press.

Willekens, F. 2009. “Continuous-Time Microsimulation in Longitudinal Analysis”. In *New Frontiers in Microsimulation Modelling*, edited by A. Zaidi, A. Harding, and P. Williamson, pp. 413–436. Ashgate.

AUTHOR BIOGRAPHIES

OLIVER REINHARDT is a Ph.D. student in the modeling and simulation group at the university of Rostock. His email address is oliver.reinhardt@uni-rostock.de.

ADELINDE M. UHRMACHER is professor at the Institute of Computer Science, University of Rostock and head of the modeling and simulation group. Her email address is adelinde.uhrmacher@uni-rostock.de.