Proceedings of the 2021 Winter Simulation Conference
S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, eds.

INFERRING DEPENDENCY GRAPHS FOR AGENT-BASED MODELS USING ASPECT-ORIENTED PROGRAMMING

Justin Noah Kreikemeyer
Till Köster
Adelinde M. Uhrmacher
Tom Warnke

Institute for Visual and Analytic Computing
University of Rostock
Albert-Einstein-Str. 22
Rostock, 18059, GERMANY

ABSTRACT

Population-based CTMC models can generally be executed efficiently with stochastic simulation algorithms (SSAs). However, the heterogeneity in agent-based models poses a challenge for SSAs. To allow for an efficient simulation, we take SSAs that exploit dependency graphs for population-based models and adapt them to agent-based models. We integrate our approach with object-oriented frameworks for agent-based simulation by detecting dependencies via aspect-oriented programming (AOP). This way, modelers can implement models without manually recording dependency information, while still executing the models with efficient, dependency-aware SSAs. We provide an open-source implementation of our approach for the framework MASON, showing significant speedups in model execution.

1 INTRODUCTION

Agent-based modeling (ABM) is used in a myriad of domains, such as biology, traffic control, social science, and operations research (Heath et al. 2009). The fundamental concept of these models is the representation of entities of the target system as interacting attributed agents. The dynamics of "virtually all" (Law 2015, p. 704) agent-based models is implemented with a fixed-increment time advance, despite the problem that it might threaten the validity of simulation results (Buss and Al Rowaei 2010; Law 2015, pp. 72f.).

One of the reasons for the dominance of a fixed-increment time advance in agent-based modeling and simulation is that rescheduling events, i.e., discrete-event simulation, can quickly become complex (Warnke et al. 2016). This manual rescheduling of events can be avoided for continuous-time Markov chains (CTMCs), in which all waiting times between discrete events are exponentially distributed. Traditionally, stochastic simulation algorithms (SSAs) are aimed at population-based rather than agent-based CTMC models (Henzinger et al. 2011; Pahle 2009). The heterogeneity in agent-based CTMC models can decrease the performance of simple SSAs, caused by the frequent need for (pseudo-)random numbers and the massive state space (Warnke et al. 2016; Reinhardt and Uhrmacher 2017).

Dependency graphs can effectively increase the performance of SSAs (Gibson and Bruck 2000; Cao et al. 2004). Reinhardt and Uhrmacher (2017) have also shown this for executing agent-based models by integrating the detection of dependencies among events into the external domain-specific modeling language ML3 (Warnke et al. 2015) where it can be exploited by the ML3 simulator. The model and state changes are analyzed to determine which future events depend on parts of the state that were modified by the last event, and only those are rescheduled. The same idea can be applied, in principle, to models that are defined in a general-purpose language as used in many frameworks for agent-based modeling, such as *RePast Simphony* (North et al. 2013) (Java), *MASON* (Luke et al. 2005) (Java), and *Mesa* (Kazil et al.

978-1-6654-3311-2/21/$31.00 ©2021 IEEE

2020) (Python). However, obtaining the dependency information is challenging in this case, as the model is not available for simple analysis.

In this paper, we propose using aspect-oriented programming (AOP) to record dependencies between events. As a proof of concept, we have implemented support for CTMC-based models and an automated dependency identification in the popular ABM framework MASON. The implementation is available under a permissive open-source license at Kreikemeyer (2021).

2 DISCRETE-EVENT SIMULATION OF AGENT-BASED CTMC MODELS

We start with some definitions of central terms and define the type of model we consider. This paper builds on Warnke et al. (2016), and more background information can be found there.

It is difficult to define the boundaries of ABM, let alone give a precise definition. As Macal and North (2009) state in a tutorial on ABM, *"there is no universal agreement on the precise definition of the term 'agent' in the context of ABMS"*. Law (2015) describes in chapter 13 of the most recent edition of his introductory book to modeling and simulation: *"[...] [W]e were very surprised to find that there was not a generally accepted definition of an agent [...]"*. Based on these two sources, we use the following definitions.

An *agent* is a component in a simulation model representing an individual or object. It might for example represent a human, an animal or an organization. It may have arbitrary *attributes*, describing its specific characteristics. An agent representing a human might for example have its age as an attribute. In addition, agents live in some kind of environment. We consider models in which agents are connected in a number of undirected, unlabelled graphs. Such graphs can, for example, encode social ties between individuals, or neighborhood relations between grid cells. Finally, the *behavior* of an agent describes the agent's *actions* in dependence of its perception of the model state. An action can be seen as a function that transforms the state of the model, for example by changing the acting agent's own or other agents' attributes, creating or removing agents, or changing the network(s).

2.1 CTMCs and SSAs

In this paper, we focus on models in which the agent's behavior can be captured in (homogeneous) continuous-time Markov chains (CTMCs). In CTMCs, the decision of when the next state transition occurs and what the successor state will be depends only on the current state, and not on the total simulation time or how long the model has been in the current state (*memorylessness*). Formally, a CTMC is a stochastic process $X(t)$ with $t \in \mathbb{R}^{\geq 0}$, where the $X(t)$ come from a state space S and for any $0 \leq s \leq t$ and $i, j \in S$

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

While this property limits the expressiveness of CTMCs, they are the foundation of many modeling formalisms, in particular for population-based modeling (Henzinger et al. 2011). These formalisms specify a CTMC by defining the transitions between states, where each state transition is equipped with a rate. In accordance with the definition of a CTMC above, a state transition occurs after an exponentially distributed waiting time, where the parameter of the distribution is the transition's rate.

Thus, CTMCs are fundamentally based on continuous time; consequently, simulation of CTMCs is driven by events at arbitrary time points (i.e., proper discrete-event simulation). This distinguishes agent-based CTMC models from the majority of agent-based models, in which events can only occur at specific equidistant points in time (Law 2015, p. 73). Such step-wise simulation does not allow for a good representation of physical time. Also, the step size has to be chosen wisely, as there is a trade-off between simulation performance and accuracy of the results. As Buss and Al Rowaei (2010) argue, the step size has to be viewed as a parameter of the model that can affect the outcome in the same way as any other parameter.

To execute CTMCs, an entire family of so-called stochastic simulation algorithms (SSAs) has been developed. The core idea of the first reaction method (FRM), which was introduced in the 70s (Doob 1945;

Gillespie 1976), is to let all possible state transitions compete in a *stochastic race* according to the current model state. For each state transition a waiting time from the associated exponential distribution is drawn. The fastest transition is executed, and the process is repeated. Subsequent SSAs refined and optimized this idea to increase the simulation efficiency. For example, the direct method (DM) and its variants only require two random numbers per simulation step (Gillespie 1977).

One influential idea is the introduction of a dependency graph, first proposed in the next reaction method (NRM) (Gibson and Bruck 2000). The next reaction method uses a scheduling approach for the reactions. After a reaction fires, the times of all affected reactions are updated based on the dependency graph. It was shown that the benefit of such a dependency graph applies to agent-based models as well (Reinhardt and Uhrmacher 2017). In an optimization of the DM, a dependency graph is used to reduce the effort in updating the sum over all individual reaction propensities, which determines the reaction and time of the next event (Cao et al. 2004). In addition to the dependency graph, this optimized direct method (ODM) uses a sorting of propensities to decrease reaction selection time.

2.2 Running example: SIRS with spontaneous infections

To illustrate the kind of models that fit the above description, we consider as an example a simple model of an infectious disease spreading in a social network. In this model, agents are connected in a network and each agent has one attribute that describes its infection status. This attribute has one of three values:

susceptible The agent is currently healthy, but not immune to the disease.
infectious The agent is currently infected and spreads the disease to its social contacts.
recovered The agent is healthy and immune to the disease.

There are four types of events in this model:

infection A susceptible agents gets infected by an infectious social contact. The rate is proportional to the number of infectious network neighbors.
recovery An infectious agent recovers after some time and is now immune to the disease.
loss of immunity A recovered agent becomes susceptible.
spontaneous infection A susceptible agent becomes infectious without having an infectious contact.

This type of model is often called an *SIRS* model after the typical sequence of states an agent will pass through (Hethcote 1976). Additionally, we introduce a spontaneous infection event. For us, this is a useful example for two reasons. First, the model never reaches a state in which no more actions are possible. Instead, it keeps on running forever, which will allow us to easily measure the steady-state performance of running the model with different simulation algorithms in section 5. Second, the model shows the impact of *locality* on the dependencies between events. In a given model state, an event will only change the attribute of a single agent. Thus, only a subset of the agents in the model will be able to observe this change—in this example, only the direct neighbors. For example, if a specific infectious agent recovers, all its susceptible neighbors now have one less infectious contact. Their behavior needs to be adapted, as the rate for the infection event has decreased. All other agents perceive no change in the model state and do not adapt their behavior.

The efficiency of simulating this model benefits from exploiting locality, as (depending on the density of the network) only a small subset of the agent population must be considered per simulation step. We now present two ways to implement this SIRS model.

3 IMPLEMENTING AGENT-BASED MODELS

We first show how an agent-based model like the SIRS model can be implemented using discrete-event simulation in an ABM framework. Here, locality can be exploited by selectively rescheduling events as part

of executing an agent's action. Second, we show how dedicated agent-based modeling languages exploit locality by using an SSA with a dependency graph.

3.1 Object-oriented Frameworks for Agent-based Modeling

In response to the popularity of ABM, a variety of ABM frameworks have been developed, for example *RePast Simphony* (North et al. 2013), *MASON* (Luke et al. 2005), and *Mesa* (Kazil et al. 2020). Those handle common implementation details, such as data structures for scheduling events and also ABM-specific modeling methods and tools, like a base class for agents (when using an object-oriented approach) or tools for visualization. Comprehensive overviews of current frameworks can be found in Railsback et al. (2006) and Abar et al. (2017). In this paper, we focus on MASON as an exemplary framework.

MASON is a framework that provides deep control over the model and the freedom to implement a model using the full capabilities of Java and object orientation. Agents are implemented as objects that communicate with each other via method calls and attribute accesses. The schedule is also an object, and agents can add events to the schedule for some future time point by calling a method on the schedule. Events are objects as well, implementing a method `step` that is invoked when the event fires. The following listing shows a simplified snippet from a MASON implementation of the SIRS model.

```
1  public class Agent {
2      InfectionState infectionState;
3      SimState model;
4
5      public void recover() {
6          this.infectionState = RECOVERED;
7          scheduleImmunityLoss();
8          /* ... trigger susceptible neighbors to reconsider their infection event ... */
9      }
10
11     private void scheduleImmunityLoss() {
12         double rate = 0.05 / 1.0;
13         double waitingTime = (1.0 / rate) * Math.log(1.0 / model.random.nextDouble());
14         model.schedule.scheduleActionOnceIn(waitingTime, new Steppable() {
15             @Override public void step(SimState model) { Agent.this.loseImmunity(); }
16         });
17     }
18 }
```

The snippet includes the method `recover` that is invoked when an infectious agent recovers. It changes the state of the agent, but is also responsible for scheduling the next event for this agent by calling the method `scheduleImmunityLoss` as well as trigger network neighbors to reschedule their infection event, if present. This illustrates how the code for an actual model as described above (i.e., types of states and events) is intertwined with the code for the simulation logic (i.e., exploiting dependencies between events). As a consequence of this lack of separation of concerns, the simulation code is not easily reusable, and the model can not easily be executed with another simulation algorithm (see Warnke et al. (2016) for more details).

3.2 Agent-based Modeling with Rules

As an alternative to using an ABM framework based on a general-purpose programming language, it is also possible to use a *domain-specific language* (DSL) (Fowler 2010). A DSL provides syntax and

semantics tailored to the problem at hand, often allowing for a more succinct and readable representation of a program. To support discrete event simulation of agent-based models, we have developed the DSL "Modeling language for linked lives" (ML3) (Warnke et al. 2015; Warnke et al. 2017). In ML3, agents and their attributes are represented similarly as in the object-oriented paradigm. Their behavior, however, is expressed with *rules* that follow the paradigm of stochastic guarded commands (Henzinger et al. 2011). The recovery of an agent can be expressed in ML3 as follows:

```
1 Agent
2 |  ego.status = "infectious"
3 @  0.05
4 -> ego.status := "recovered"
```

Here, the second line is a *guard*, constraining which agents this rule applies to, and the third and fourth line denote the rate and the effect of the action.

An ML3 *simulator* reads in such a model description and executes it using an SSA, which is independent from a concrete model and, thus, reusable across models. As the simulator is responsible for evaluating guards and rates as well as executing actions (i.e., applying changes to the model state), it can track which events affect each other and reschedule events accordingly (Reinhardt and Uhrmacher 2017). Thus, the modeler is relieved from exploiting locality manually.

Although tailored DSLs such as ML3 have the advantages mentioned above, they also have some disadvantages when compared to ABM frameworks. On the implementer's side, it is necessary to implement what is essentially a complete programming language. ML3, for example, provides local variables and user-defined functions, whose semantics must be encoded in the simulator. Still, ML3 supports no structured data types apart from sets to encode the knowledge of agents. If more complex knowledge bases are required by the application, this implies a less succinct description and a less efficient execution of models (Reinhardt et al. 2019). However, this is a trade-off to avoid re-implementing many common features of general purpose languages, such as type inference.

3.3 Integrating Rules with Object-oriented Frameworks

To combine the advantages of ABM frameworks and DSLs, we have proposed integrating a DSL for agent-based CTMC modeling into ABM frameworks (Warnke et al. 2016). The key idea here is not to use an *external* DSL like ML3 (i.e., a DSL with standalone syntax and semantics), but rather an *internal* DSL (i.e., a DSL that reuses the syntax and semantics of an existing language) (Fowler 2010). This allows integrating the rule paradigm into the object-oriented agent implementation of ABM frameworks (Figure 1). For example, the recovery rule can be expressed as follows:

```
1 public class Agent {
2     InfectionState infectionState;
3
4     this.addAction(() -> this.infectionState == INFECTIOUS,
5                    () -> 0.05,
6                    () -> this.infectionState = RECOVERED));
7 }
```

We use lambda expressions to encode the guard, the rate, and the effect of actions. In each of those, the attributes of an agent (or other agents accessible via network edges) can be accessed, and complex behavior can be implemented with the full expressive power of Java. To execute the model, an intermediate layer evaluates all rules for all agents and schedules events in the ABM framework's schedule. Thus, the model can be executed like any other model implemented in the ABM framework, and all tooling can be

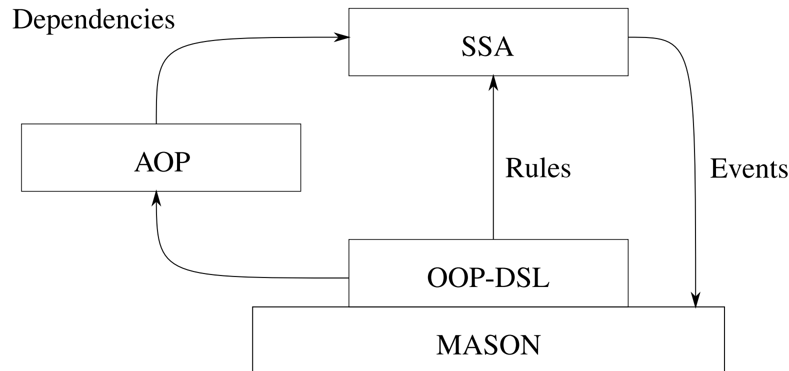

Figure 1: High-level overview of our approach. The SSA evaluates the rules expressed in the OOP-DSL for all agents and schedules events in MASON accordingly. Our main contribution is the addition of AOP on the left, which allows the automatic tracking of dependencies between actions, enabling the realization of efficient model-independent SSAs for agent-based models.

applied to it. However, one advantage of the external DSL implementation is lost: As rules are expressed in plain Java code, there is no component that is able to supervise which agents are affected by an action. Therefore, the locality of actions can not directly be exploited. In the following, we present an approach to recover the ability to exploit locality by augmenting our approach with aspect-oriented programming.

4 TRACKING DEPENDENCIES WITH ASPECT-ORIENTED PROGRAMMING

AOP is a programming paradigm that was designed to separate cross-cutting code from the modules it cuts across (Kiczales et al. 1997). It is based on three concepts: join points, point cuts and advice, where the last two form an aspect. An aspect provides the ability to define cross-cutting code as snippets, called *advice*. These advice might apply at various places in the program flow, called *join points*. *Point cuts*, which act as filters for join points, can be used to select all join points an advice should be applied to. Through a process called weaving, the advice (i.e., the cross-cutting logic) are conjoined with the program at compile time. Thus, they can be completely separated from the program at the time of definition. In this case, this is exploited to weave the code for recognizing dependencies into the model after it was defined (Figure 1).

AspectJ™ is an implementation of the AOP paradigm for Java (The Eclipse Foundation 2021; Team 2002). It extends the language with syntax and tools to define aspects in a similar way to classes: They can be defined in their own file using the `aspect`-keyword. Within an aspect definition, it is possible to create named point cuts for method calls (`call`) and executions (`execution`), operations on fields (`get`,`set`) and also the `initialization` of classes, to name a few. Advice can be targeted `before`, `after` or `around` the join points picked by the corresponding point cut. Beyond that, they can also use certain parts of the dynamic program state, such as the current parameters of a method call, which can be retrieved using `args`, or information on the calling/targeted object, which can be accessed with `this` and `target`. It is also possible to use wild cards, such as $*$ (any string) and + (any subclass) to further generalize a point cut.

Listing 1 shows an example aspect, which uses these concepts to track and maintain attribute dependencies. The named point cut `AttributeAccess` in line 3 applies to all join points where a public field of an agent subclass is accessed and exposes the owner of the field to the advice. The `after` advice in line 6 constructs the initial dependency graph by adding the resulting dependency relations to the graph, using the signature of the join point as attribute name. In the SIRS model, this would add a dependency relation between actions and the infection states they access (i.e., depend on). Of course this advice should only apply in certain control flows, such as the initialization of the NRM where every action is evaluated once, but for simplicity, those restrictions are omitted here.

Listing 1: Simplified example aspect for discovering, maintaining and exploiting attribute dependencies. Any public field of an agent-subclass is considered an attribute. In the SIRS model, the only attribute of agents would be their infection state.

```
1  public aspect DependencyAspect {
2
3      pointcut AttributeAccess(Agent owner):
4          get(public * Agent+.*) && target(owner);
5
6      after: AttributeAccess(Agent owner) {
7          nrm.attributeDeps.addDep(currentAction, thisJoinPointStaticPart.getSignature(), owner);
8      }
9
10     after(Agent calledOn): AttributeMutation(calledOn) {
11         String attr = jp.getSignature();
12         HashSet<Action> deps = nrm.attributeDeps.GetDependents(attr, calledOn);
13         toReschedule.addAll(deps);
14     }
15
16     /* After a step of the NRM concluded. */
17     after(NextReactionMethod nrm, Action currentAction) returning: Step(nrm, currentAction) {
18         /* ... handle removed agents and edges ... */
19         for (Action dep : toReschedule) {
20                 /* If the owner of this action was removed, do not reschedule it. */
21                 if (!removedAgents.contains(dep.getOwner())) {
22                         nrm.attributeDeps.RemoveDeps(dep);
23                         nrm.edgeDeps.RemoveDeps(dep);
24                         nrm.rescheduleAction(dep, currentAction);
25                 }
26         }
27         /* ... handle new agents ... */
28     }
29 }
```

The discovered dependency relations are put to use every time an attribute is altered to reevaluate the guard and rate of the depending actions. To achieve this, another advice defined in line 10 is applied to every *attribute mutation*, which adds the corresponding dependents to a list of actions to be updated. Hence, it records in each step which scheduled events need to be updated. They are not updated directly, as in actions with multiple mutations they might be updated more than once. Also, in some scenarios the order of the updates is important. For example, a removed agent must not be updated. Using the deferred update allows actions to always work with the current state instead of a partially updated one. Thus, `after` the execution of an action concluded, a third advice (line 17) applies, which handles the updates themselves: It iterates over all actions in the list and updates them. During the update, an advice similar to the first also applies again to discover any changes in the dependencies resulting from the mutation.

Besides the mutation of attributes, it is also necessary to track *edge additions*, *edge removals*, *agent additions* and *agent removals*. The above approach can analogously be applied to discovering and using dependencies on edges of the network, which are needed for handling mutations of the network. One special case is the addition of an arbitrary edge. In this case, all participants of the new edge, as well as all

actions depending on one of the adjacent edges need to be updated. This is because their possible access paths to other agents might be completed by the addition and thus it may unveil a new attribute dependency. As the dependency tracking is quite complex, not all the details are presented here due to spatial limitations. For details, we refer to the `DependencySymbiont` aspects of the implementation (Kreikemeyer 2021).

5 EVALUATION

We evaluate our aspect-oriented method for dependency-tracking on the SIRS model introduced in subsection 2.2. Example outputs of this model for the tested SSAs are provided in Figure 2. We compare the following algorithms:

- A *manual* implementation of the dependency tracking, as presented in subsection 3.1.
- An implementation of the next reaction method that uses aspect-oriented programming to automatically derive the dependencies (*AO-NRM*).
- An implementation of the direct method that also uses an automatically derived dependency graph to update the sum of rates (*AO-DM*).
- An implementation of the first reaction method (*FRM*). We chose this as a representative of the basic methods that do not exploit dependencies. We also implemented the direct method, which performs similarly.

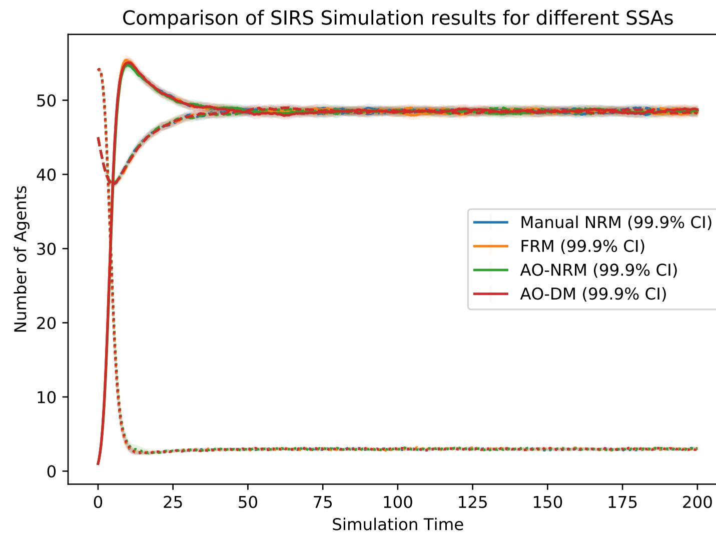

Figure 2: Example output curves of different SSAs on the SIRS model. Shown is the mean over 800 replications of the population of susceptible (dotted line), infectious (solid line) and recovered (dashed line) agents. The faint shadows show the 99.9% confidence intervals.

The performance is analyzed using the Java Microbenchmark Harness (JMH) (Oracle 2021) to measure the throughput in agent-actions (events) per second. For every parameter configuration, 3 Java Virtual Machine (JVM) forks of 10 warm-up iterations of 1 second and 20 measurement iterations of 2 seconds each are performed. This counteracts the fluctuations commonly observed with the JVM. The implementation (Kreikemeyer 2021) includes instructions and a script to reproduce the results.

Figure 3 shows the steady-state throughput of the SIRS model using a grid-like contact network for different population sizes, where each agent has its top, right, bottom and left neighbors as contacts. It is evident that for such a low mean degree of the graph (≈ 4), the dependency-based methods have a clear advantage over the other methods, which increases with the size of the population. However, among the

dependency-based methods, the manual NRM has a constant advantage of around one order of magnitude over the AO-NRM. This results from the overhead of maintaining the dependencies, which only the latter relies on. We can also observe the known fact that the direct method does not scale well for a large number of reactions. In the limit of many reactions, time per step is $\mathcal{O}(n)$, in the same way, the FRM is $\mathcal{O}(n)$.

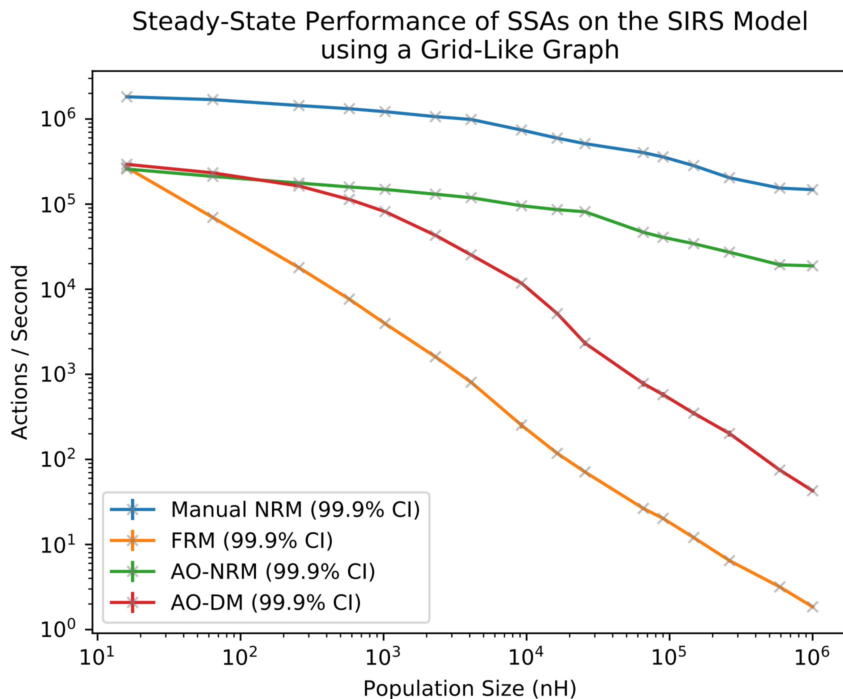

Figure 3: Comparison of the steady-state performance of the SIRS model on a grid-like contact network with constant mean degree of 4.

The constant mean degree of the grid-like contact network provides excellent conditions for the dependency-based methods' optimization. However, it is important to also consider the density of the network. Figure 4 shows the steady-state throughput of the SIRS model on an Erdős-Rényi contact network with varying density. This indicates a limitation of the aspect-oriented methods, as with increasing density of the contact network, their overhead for maintaining the dependencies increases, whereas (in this case) the FRM is not influenced. The manual NRM performs best, as it has no overhead and still exploits the locality of events.

In contrast to the FRM, the dependency-based methods require an initialization in order to determine the initial dependency graph and possibly the initial events (NRM). This overhead is not accounted for in the steady-state performance shown previously. Figure 5 characterizes the average time taken for the initialization of the dependency-based methods and compares it to the first step of the FRM. As this time can be very short, the parameters for the JMH benchmarks are adjusted to measure the single shot time over 5 forks of 10 warm-up and 40 measurement iterations. The results show that the aspect-oriented SSAs have a clear disadvantage over the other SSAs. In the general case, this overhead should lose importance with the length of the replications.

6 CONCLUSION

Internal DSLs, realized as a layer on top of an agent-based modeling framework, allow a succinct description of continuous-time agent-based models and a clear separation of concerns between model and simulation.

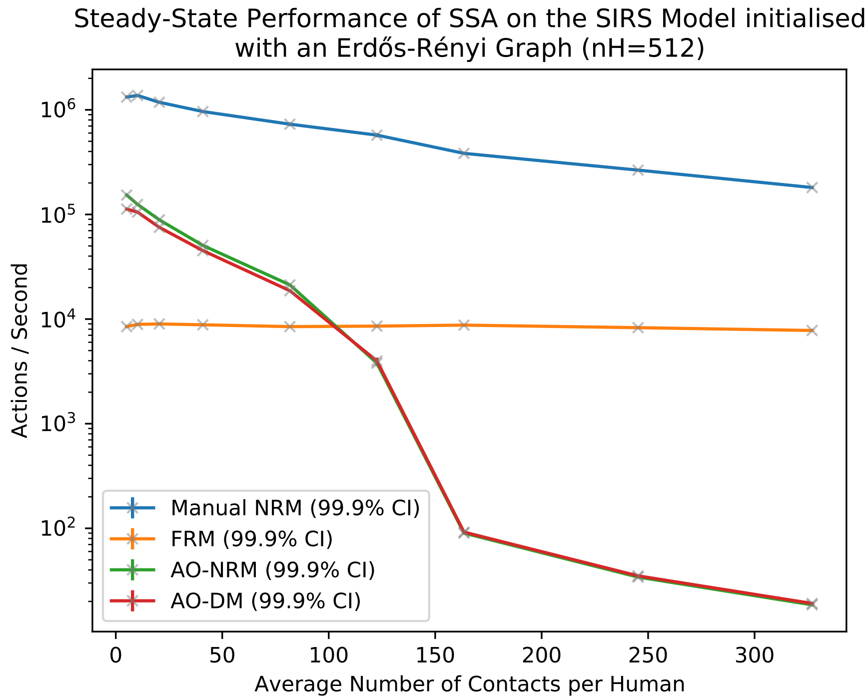

Figure 4: Comparison of the steady-state performance of the SIRS model on an Erdős Rényi graph with increasing density. The population (nH) is constant.

An efficient simulation of these models relies on effectively tracking the dependencies between agents' activities. In this paper we explored the use of aspect-oriented programming (AOP) to record dependencies between events. As a proof of concept, we implemented support for CTMC-based models and an automated dependency identification in the popular ABM framework MASON. With AOP, we can separate the user-written model from the model-independent component for dependency tracking. This gives the modeler the freedom to use the entire feature set of the implementation language and the ABM framework, while separating the concerns of efficient simulation.

The performance results show that AOP is a suitable method to integrate dependency tracking into a model description written in a general-purpose programming language. The key for getting this to work is the object-oriented rule interface, given as an internal DSL. First, this provides the theoretical underpinning for interpreting the agent-based model as a CTMC, and thus allowed us to use established SSAs, such as the NRM and DM, to run discrete-event simulations. Second, the rule interface structures the model definition in a way that allows us to define point cuts to precisely capture the evaluation of rules and execution of actions via AOP. Third, the rule interface establishes the connection between the underlying ABM framework MASON and our implementation, which implies that our approach is usable for any ABM framework onto which such a rule interface can be layered (e.g., we used Repast Simphony in Warnke et al. (2016)).

Whereas our AOP-based implementations are outperformed by the hand-crafted SIRS model from subsection 3.1, they still show the speed-up caused by exploiting locality. In contrast to the hand-crafted implementation, they are also reusable across models and lead to a much cleaner, more readable model description. Moreover, the aspect source code is an expressive description of tracking dependencies and actions during simulation, facilitating easy modification and extension.

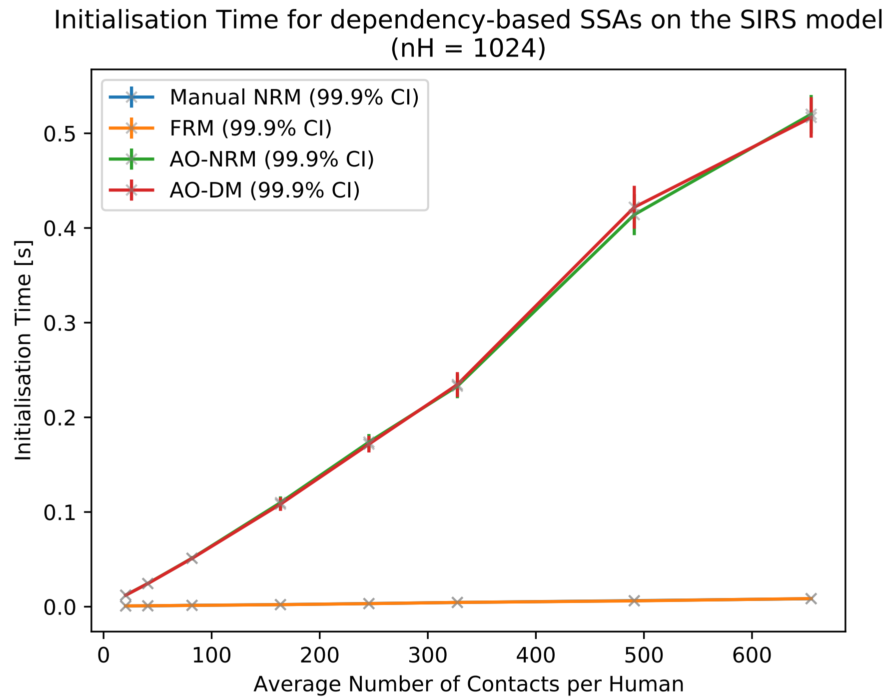

Figure 5: Initialization overhead of the dependency-based SSAs under a constant population (nH). The FRM and manual NRM overlap.

Still, it may be promising to explore techniques such as macros or code generation to integrate tracking of dependencies directly into the DSL implementation. The concrete choices depend on what is available in the ecosystem of the respective programming language.

ACKNOWLEDGEMENTS

Financial support was provided by the Deutsche Forschungsgemeinschaft (DFG) research grant ESCeMMo (UH-66/13).

REFERENCES

Abar, S., G. K. Theodoropoulos, P. Lemarinier, and G. M. O'Hare. 2017. "Agent Based Modelling and Simulation tools: A review of the state-of-art software". *Computer Science Review* 24:13–33.

Buss, A., and A. Al Rowaei. 2010. "A comparison of the accuracy of discrete event and discrete time". In *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, 1468–1477. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Cao, Y., H. Li, and L. Petzold. 2004. "Efficient formulation of the stochastic simulation algorithm for chemically reacting systems". *The Journal of Chemical Physics* 121(9):4059–4067.

Doob, J. L. 1945. "Markoff Chains–Denumerable Case". *Transactions of the American Mathematical Society* 58(3):455–473.

Fowler, M. 2010. *Domain-specific languages*. Upper Saddle River, NJ, USA: Pearson Education.

Gibson, M. A., and J. Bruck. 2000. "Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels". *The Journal of Physical Chemistry A* 104(9):1876–1889.

Gillespie, D. T. 1976. "A general method for numerically simulating the stochastic time evolution of coupled chemical reactions". *Journal of Computational Physics* 22(4):403–434.

Gillespie, D. T. 1977. "Exact stochastic simulation of coupled reactions". *Journal of Physical Chemistry* 81(25):2340–2361.

Heath, B., R. Hill, and F. Ciarallo. 2009. "A Survey of Agent-Based Modeling Practices (January 1998 to July 2008)". *Journal of Artificial Societies and Social Simulation* 12(4):9.

Henzinger, T., B. Jobstmann, and V. Wolf. 2011. "Formalisms for specifying Markovian population models". *International Journal of Foundations of Computer Science* 22(04):823–841.

Hethcote, H. W. 1976. "Qualitative analyses of communicable disease models". *Mathematical Biosciences* 28(3-4):335–356.

Kazil, J., D. Masad, and A. Crooks. 2020. "Utilizing Python for Agent-Based Modeling: The Mesa Framework". In *Social, Cultural, and Behavioral Modeling*, edited by R. Thomson, H. Bisgin, C. Dancy, A. Hyder, and M. Hussain, 308–317. Cham, CH: Springer International Publishing.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. 1997. "Aspect-oriented programming". In *ECOOP'97 — Object-Oriented Programming*, edited by M. Akşit and S. Matsuoka, 220–242. Berlin, Heidelberg: Springer.

Kreikemeyer, Justin Noah 2021. "AO-ABM-SSA". https://git.informatik.uni-rostock.de/mosi/ao-abm-ssa.

Law, A. M. 2015. *Simulation Modeling and Analysis*. 5th ed. New York, NY, USA: McGraw-Hill Education.

Luke, S., C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. 2005. "Mason: A Multi-Agent Simulation Toolkit". *SIMULATION* 81(7):517–527.

Macal, C. M., and M. J. North. 2009. "Agent-based modelling and simulation". In *Proceedings of the 2009 Winter Simulation Conference*, edited by M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, 86–98. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

North, M. J., N. T. Collier, J. Ozik, E. Tatara, E. Altaweel, C. M. Macal, M. Bragen, and P. Sydelko. 2013. "Complex Adaptive Systems Modeling with Repast Simphony". *Complex Adaptive Systems Modeling* 1:3.

Oracle 2021. "OpenJDK: jmh". https://openjdk.java.net/projects/code-tools/jmh/. Accessed 5th August 2021.

Pahle, J. 2009. "Biochemical simulations: stochastic, approximate stochastic and hybrid approaches". *Briefings in Bioinformatics* 10(1):53–64.

Railsback, S. F., S. L. Lytinen, and S. K. Jackson. 2006. "Agent-based Simulation Platforms: Review and Development Recommendations". *SIMULATION* 82(9):609–623.

Reinhardt, O., M. Hinsch, J. Bijak, and A. M. Uhrmacher. 2019. "Developing Agent-Based Migration Models in Pairs". In *Proceedings of the 2019 Winter Simulation Conference*, edited by N. Mustafee, K.-H. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, 2713–2724. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Reinhardt, O., and A. M. Uhrmacher. 2017. "An efficient simulation algorithm for continuous-time agent-based linked lives models". In *Proceedings of the 50th Annual Simulation Symposium*, edited by S. Jafer et al. San Diego, CA, USA: Society for Computer Simulation International.

AspectJ Team 2002. "The AspectJ™ Programming Guide". https://www.eclipse.org/aspectj/doc/released/progguide/index.html. Accessed 5th August 2021.

The Eclipse Foundation 2021. "The AspectJ Project". https://www.eclipse.org/aspectj/. Accessed 5th August 2021.

Warnke, T., O. Reinhardt, A. Klabunde, F. Willekens, and A. M. Uhrmacher. 2017. "Modelling and simulating decision processes of linked lives: An approach based on concurrent processes and stochastic race". *Population studies* 71(sup1):69–83.

Warnke, T., O. Reinhardt, and A. M. Uhrmacher. 2016. "Population-based CTMCS and agent-based models". In *Proceedings of the 2016 Winter Simulation Conference*, edited by T. M. K. Roeder, P. I. Frazier, R. Szechtman, E. Zhou, T. Huschka, and S. E. Chick, 1253–1264. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

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*, edited by L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, 2764–2775. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

AUTHOR BIOGRAPHIES

JUSTIN NOAH KREIKEMEYER is a student in the Computer Science Master's program at the University of Rostock. His e-mail address is justin.kreikemeyer@uni-rostock.de.

TILL KÖSTER is a doctoral researcher in the modeling and simulation group at the Institute for Visual and Analytic Computing, University of Rostock. His e-mail address is till.koester@uni-rostock.de.

ADELINDE M. UHRMACHER is a professor at the Institute for Visual and Analytic Computing, University of Rostock, and head of the modeling and simulation group. Her e-mail address is adelinde.uhrmacher@uni-rostock.de.

TOM WARNKE is a postdoctoral researcher in the modeling and simulation group at the Institute for Visual and Analytic Computing, University of Rostock. His e-mail address is tom.warnke@uni-rostock.de.