An Agent-Based Simulation API for Speculative PDES Runtime Environments

Andrea Piccione
piccione.1422045@studenti.uniroma1.it
Sapienza, University of Rome
Rome, Italy

Matteo Principe
matteo.principe@students.uniroma2.eu
University of Rome "Tor Vergata"
Rome, Italy

Alessandro Pellegrini
pellegrini@diag.uniroma1.it
Sapienza, University of Rome
Rome, Italy

Francesco Quaglia
francesco.quaglia@uniroma2.it
University of Rome "Tor Vergata"
Rome, Italy

ABSTRACT

Agent-Based Modeling and Simulation (ABMS) is an effective paradigm to model systems exhibiting complex interactions, also with the goal of studying the emergent behavior of these systems. While ABMS has been effectively used in many disciplines, many successful models are still run only sequentially. Relying on simple and easy-to-use languages such as NetLogo limits the possibility to benefit from more effective runtime paradigms, such as speculative Parallel Discrete Event Simulation (PDES). In this paper, we discuss a semantically-rich API allowing to implement Agent-Based Models in a simple and effective way. We also describe the critical points which should be taken into account to implement this API in a speculative PDES environment, to scale up simulations on distributed massively-parallel clusters. We present an experimental assessment showing how our proposal allows to implement complicated interactions with a reduced complexity, while delivering a non-negligible performance increase.

CCS CONCEPTS

• **Computing methodologies → Discrete-event simulation**; **Agent / discrete models**; **Simulation support systems**; *Massively parallel and high-performance simulations.*

KEYWORDS

agent based modeling and simulation, parallel discrete event simulation, runtime environments

ACM Reference Format:
Andrea Piccione, Matteo Principe, Alessandro Pellegrini, and Francesco Quaglia. 2019. An Agent-Based Simulation API for Speculative PDES Runtime Environments. In *SIGSIM Principles of Advanced Discrete Simulation (SIGSIM-PADS '19), June 3–5, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3316480.3322890

1 INTRODUCTION

Agent-Based Modeling and Simulation (ABMS) is a powerful paradigm in which the system is represented by a collection of autonomous decision-making entities (the agents) which are set out in an environment [3, 30]. Each agent individually assesses the surrounding environment, also taking into account the presence of other agents, and makes decisions on the basis of a certain set of rules which implement their behavior. During its lifetime, an agent can decide to change its behavior, also depending on the environment state and interactions with other agents. The actions that agents take might also have effects on other agents and/or on the surrounding environment—for example, an agent can *produce*, *consume*, or *exchange* items.

ABMS is considered incredibly powerful for multiple applications and real-world business problems for a number of reasons. First of all, the model developer can concentrate on the design of agents behavior independently of *where* the agents will act. This significantly simplifies the development of complex models, allowing to reach results which could be difficult when relying on more traditional mathematical methods [11, 15]. Second, the interaction of multiple agents in a system can exhibit complex behavioral patterns [39], able also to show (or even anticipate) what is commonly referred to as *emergent behavior*. Several approaches have also coupled sophisticated models with neural networks [19], evolutionary algorithms [29], or other learning techniques in order to provide the agents with behavioral adaptation, making ABMS even more powerful and realistic.

In general, ABMS requires three common aspects to be dealt with to be effectively used to find a solution for real-world problems:

- *Agent specification*: agents are described in terms *simulation state*, namely the set of explanatory variables which keep track of their evolution and interaction with the environment/other agents; this state might drive the behavior of agents during the simulation. Each agent must be identified uniquely in the whole system.
- *Interaction definition*: agents interact with the environment and with other agents. The decisions which agents make might depend also on the state of portions of the environment which are nearby the current position of the agent, or on the presence/absence of other specific agents in the (surrounding) environment.

- *Topology specification*: ABMS is extremely versatile, therefore requiring to model complex topologies. While many approaches rely on grid-based environments, more complex (graph-based) topologies can enable more interesting interactions to be described or to emerge. Modern ABMS requires to overcome the traditional definition of a physical environment, because it might be defined by several layers of (also virtual) information, and might therefore be also modified by the agents themselves.

The size of problems tackled through ABMS is growing in size at an unprecedented pace. The intrinsic dynamic nature of real-world phenomena commonly modeled through the ABMS formalism can easily lead to a significantly increased complexity due to the following reasons (also in combination):

(1) A very large number of agents are involved, or the environment is extremely large. Here, unfeasibility can materialize either in too long execution times, or in non-sufficient memory capacity on single nodes (the so-called *memory wall* [31]). This is a fundamental aspect especially when large models are necessary to disclose an emergent behavior which a small-scale variant is not able to show, e.g. when the emergent property is linked to the amount of interactions between the agents [32, 41].
(2) Increased amount of data which enable the implementation of micro-level simulation models [34], based on the inclusion of an always larger set of explanatory variables in the state of single agents.
(3) Enriched decision-making policies in the agents, such as rational behavior or cognitive and psychological processes. These aspects (mostly related to micro-simulations) can require higher computing demands.
(4) Interest in systems which exhibit a non-linear, dynamic behavior, with high uncertainty and notable degree of stochasticity. In this context, a single run of the simulation model is not sufficient: to obtain statistically-significant results, a calibration based on multiple repeated simulations is the only solution to deal with parameter sweep. While a single simulation could be feasible, a large number of simulations might be not.

While the formalism allows to easily deal with this increased complexity, traditional frameworks to run ABMS might experience performance penalties which make finding solutions to these *what-if* analysis problems unfeasible, preventing researches to find the needed insights [2]. As an example, in contexts such as demography [34], ABMS applications are developed mostly relying on sequentially-executed languages such as NetLogo [44]. This is due to the fact that developing sequential models is easy, especially for experts coming from domains not related to computer science. This is a problem so hot that researchers are investigating multiple approaches to speedup the execution of agent-based models on extremely parallel architectures [48].

From an architectural point of view, Discrete-Event Simulation (DES) is the paradigm which best fits the runtime requirements of ABMS, since the impulsive nature of the events proper of DES perfectly embraces the way agent-based models should be executed. To meet the current scale of these models, the literature on Parallel DES (PDES) [17] has identified in the speculative Time Warp synchronization protocol [23] a viable solution to cope with large-scale parallelism.

Anyhow, when dealing with speculative PDES, the modeler might be exposed to many details which should not be mastered by domain experts. This is strictly related to the fact that Time Warp enforces consistency by means of the *rollback operation*. Any time that it is determined that the execution has reached an inconsistent point in the simulation, e.g. due to the reception of an out-of-order event, a previous consistent snapshot is identified and restored, allowing the trajectory to deviate and resolve the causality violation. The rollback operation has been traditionally implemented either by means of state saving and restore (since the seminal paper in [23]), or by means of reverse computation [6]. In the former case, the simulation model developer is often exposed to the state saving operation, either in terms of the invocation of explicit services exposed by the runtime environment to take a checkpoint, or by the need to annotate or notify the runtime environment of the organization in memory of the simulation state. In the latter case, the modeler is often asked to implement by hand *reverse events*, which might even double the time and cost of model development, although some proposals in the literature have tried to automatize this process (see, e.g., [8, 26, 46]).

When dealing with the large number of domains which benefit from ABMS, it is simply not possible to demand from the developers to explicitly tackle all the aforementioned technical aspects. Therefore, in this paper we describe an API explicitly targeting the development of agent-based models, which has been devised keeping in mind the three aforementioned needs of ABMS (namely, agent specification, interaction definition, and topology specification). This API is meant to be semantically-rich, thus allowing domain experts to benefit from it in the process of defining an agent-based model. At the same time, it has been designed also keeping in mind the peculiarities of speculative PDES run on distributed clusters of massively-parallel machines, hiding away from the modeler its complexity. We have considered generic runtime environments when devising the API[1], focusing anyhow on multi-threaded simulation kernels. Indications to implement this API in speculative runtime environments are described in this paper in an attempt at bridging the gap between the users of ABMS runtime environments, and the architectural capabilities of modern computing infrastructures. We evaluate both the effects on programmability, and the performance of an implementation of the API. Moreover, this API can be regarded also as the target of other approaches to simplify the programming model, e.g. the use of domain-specific languages (such as OpenABL [13]) which could be effectively transpiled to our API.

The remainder of this paper is organized as follows. In Section 2 we discuss related work. Section 3 presents the API specification for last-generation PDES runtime environments. An assessment of our proposal, both in terms of effects on programmability and performance of an implementation of the API is discussed in Section 4.

[1]All models and the implementation of the API are available online in the official ROOT-Sim repository, at https://github.com/HPDCS/ROOT-Sim. The artifacts used for the reproducibility assessment can be found at https://doi.org/10.5281/zenodo.2597110.

2 RELATED WORK

ABMS and PDES are two topics which have been extensively studied in the literature, and there have already been attempts to bridge the gap between the two worlds. We can find a large number of frameworks to support agent-based simulation in the literature. Some of the most known ones are NetLogo [44], Mason [28], RepastHPC [10], Swarm [22], JAMES II [20]. Many of these proposals might not be well suited for future exascale computing infrastructures. Therefore, some proposals have been made to devise frameworks which can significantly scale up—see, e.g., [27, 43], with the latter proposal also addressing model interoperability, thanks to the reliance on the High-Level Architecture. For a thorough comparison of these (and many more) ABMS frameworks, we refer the reader to the comprehensive work in [1]. Here, we discuss only proposals which have the closest relation to our contribution.

MASON [28] pays special attention to the performance of simulation execution, addressing computing-intensive models (i.e., scenarios with many agents), along with portability and reproducibility of the results across different hardware architectures. A parallel/distributed version (D-MASON) has been presented in [12], which relies on time-stepped synchronization and on the master/slave paradigm. We similarly address the performance of agent-based simulation execution, yet we do this for the case of speculative asynchronous (non-time-stepped) PDES. In particular, we benefit from the performance improvement which can stem from the Time Warp synchronization protocol, while allowing a simple implementation of agent-based models via an expressive API.

Pandora [47] is a C++-based simulation framework enabling executions in parallel/distributed environments. It features several AI algorithms for supporting agents' decision making and provides python bindings (which is a benefit for inexperienced programmers). At the same time, Pandora does not hide its internal complexity by design, allowing (and sometimes requiring) the model developer to extend its fundamental classes, just to perform simple actions. Conversely, in our proposal we offer the simulation model developer an API that is specifically tailored for implementing agent-based models, and which hides away all the idiosyncrasies of synchronization in a distributed simulation. This allows for a simplified implementation of simulation models, giving transparent access to highly optimized synchronization facilities to support efficient computations on clusters of modern parallel machines.

AnyLogic [4] is a commercial multi-method general-purpose simulation modeling and execution framework which can run simulations also in the cloud. AnyLogic allows users to spawn multiple simulations in parallel thanks to cloud resource provisioning. Recent versions of AnyLogic allow to deal with arbitrary topologies as we do, making it more evident the importance of this aspect for modern simulation platforms. Anyhow, the ultimate goal of the AnyLogic framework is to scale out simulations, while our proposal is intended particularly to increase the performance of single simulation runs.

FLAME [21] is a simulation framework targeting large, complex models with large agent populations to be run on HPC platforms using MPI and OpenMP. The counterpart FLAME GPU [40] targets 3D simulations of complex systems with a multi-massive number of agents on GPU devices. We keep the ability to deal with large number of agents, yet we rely on traditional CPU-based execution of the simulation model.

RepastHPC [10] and Swarm [22] are two ABMS runtime environments which have been successfully used to deliver high performance of agent-based models. These runtime environments support different programming languages, and allow agents to interact through the exchange of discrete events. Differently from our proposal, they require high programming skills to be effectively used. Therefore, they are commonly regarded as complex-to-use frameworks [1].

RAMSES [7] is an ABMS runtime environment, with a focus on transparency. An ABMS API has been already proposed in [7], with a goal similar in spirit to that of our proposal. The main differences between the two works are that: i) in [7], the API is based on the implementation of complex functions which are passed via pointers to the API, making it difficult to create bindings in different languages; ii) RAMSES supports only reversible computation, while the API which we propose can be implemented in both reverse computation-based and state saving-based runtime environments; iii) if an agent has to make decisions based on the state of the surrounding environment, this has to be implemented via explicit message passing; iv) dynamic topologies are not supported.

With respect to programmability, OpenABL [13] is a recent proposal of a domain-specific language which allows to formulate agent-based models in a way which is independent of the actual hardware on which the simulations should be run. While the benefit of relying on domain-specific languages is undoubted to simplify the development process of a simulation model, in this paper we focus on runtime environments. In this sense, our proposal is complementary to that of OpenABL.

3 THE ABMS API

As mentioned, we propose an API for ABMS which is semantically-rich for the developers and which, at the same time, targets runtime implementations to transparently bridge the gap towards efficient speculative PDES runs. We organize the presentation of the API functions depending on the specific aspects which they tackle.

3.1 Modeling Agents and their Interactions

We describe here all the functions exposed by the API, also describing what are the aspects that an implementation within any (speculative) runtime environment should take into account to ensure performance and consistency of the simulation. The API is presented as signatures of a possible C implementation, but by no means it is limited to this programming language. As a preliminary note, this is a stateful API: therefore, the runtime environment must maintain a set of data structures to keep track of the evolution of the simulation. For the sake of performance, we suggest that these data structures are setup lazily, i.e. when a first call to the API requiring them is issued.

Taking inspiration from [7, 36], and taking also into account the way agent-based models are commonly implemented, in our programming model we map environmental regions to LPs, while agents are mapped to data structures (or objects, in runtime environments supporting object-oriented programming). Agents are therefore handled as variables, a simple concept which is mastered also by non-experienced programmers from a multitude of domains. This is a fundamental choice, because it can significantly simplify the implementation of multiple agents' interactions in a neighboring region. Moreover, although we talk of the environment, by no means ABMS is limited to physical environments. Therefore, it is possible that the "regions" of the environment represent *logical entities*, and this aspect must be taken into account by the runtime environment—this is exactly why we do not force developers to be stuck, e.g., to grid-based environments, but we support also generic (graph-based) environments—for the sake of simplicity of presentation, we will talk of *regions* in the remainder of the paper, also when referring to a node in an abstract graph topology.

The first aspect to take into account is how to identify an agent. To this end, we introduce the `agent_t` type, which can be regarded as a *unique identifier* (UID) describing a single agent system-wide. The runtime environment should be able, also in a distributed setting, to generate `agent_ts` uniquely. Each simulation kernel instance is already commonly identified uniquely in the system (e.g., thanks to an MPI rank). Therefore, it is sufficient to rely on a per-instance monotonic counter and generate UIDs by computing the *Cantor pairing function*:

$$\frac{(K+c)\cdot(K+c+1)}{2}+c \tag{1}$$

where K is the id of the simulation kernel instance (e.g., its MPI rank) and c is the current value of the per-instance monotonic counter. This approach is quite effective—the pairing function is fast, as it relies only on integer arithmetic—and can work mostly out of the box in a multithreaded environment thanks to hardware synchronization facilities such as the *fetch and add* read-modify-write instruction[2], which allows all threads to consistently read and increment c also in a concurrent execution[3].

An efficient management of the `agent_t` type is fundamental, because agents are dynamic entities. As we will show, agents can have a lifetime shorter than that of the whole simulation, as we explicitly allow new agents to be spawned or removed from the simulation at any time. Therefore, any time that the model wants to manage agents, this must be done via API functions which deal with dynamically-handled `agent_ts`. Therefore, we suggest organizing `agent_ts` hosted within a certain portion of the environment (an LP) into a hash map, so as to reduce the latency of retrieving the data structures used to represent an agent. It is important to note that, as mentioned, the state of an agent must keep all the explanatory variables that allow the agent's logic to take decisions or change its own behavior. To this end, the simulation model developer can rely on any arbitrarily-complex data structure, and associate that at runtime with an `agent_t`. Since in an agent-based model there could be different families of agents—in the limit case, each agent might be described by a different simulation state—the simulation model developer must be allowed to easily define a per-agent data structure.

3.1.1 Basic Agent Management. After these considerations, we can now start to describe the part of the API which allows to manage the agents in the model. The following API functions are defined.

[2]This is a hardware facility which is available on all commodity CPU implementations.
[3]This approach can of course return two equal values when the counter overflows, but relying on 64-bit integers will significantly reduce this probability.

`AddAgent(unsigned user_data_size).` This is the API call which allows, at runtime, to create a new agent in the system. It returns an `agent_t` uniquely identifying the new agent system-wide. This function instructs the runtime environment to allocate and associate with the returned `agent_t` a buffer of at least `user_data_size` bytes, so as to allow the model developer to store and carry around data. Given the speculative nature of the runtime environment which we envisage, the creation of an agent should be rollbackable. In particular, this creation should be rolled back if the event during whose execution the invocation to `AddAgent()` took place is undone. Therefore, the runtime environment must associate the (dynamically-allocated) buffers to keep the agent representation with the event which caused its allocation and the LP at which this event has been executed.

The literature already offers multiple solutions which can be used to solve this problem (see, e.g., [37, 45]). In particular, if the runtime environment allows the simulation model to rely on dynamic memory (e.g., by supporting calls to `malloc()`/`free()` which are transparently rolled back), then it is sufficient to redirect the allocation of the agent data structure to per-LP memory. This means that, for a limited time span, the agent belongs to the simulation state of the LP modeling the region within which the agent is residing. This is clearly desirable, since all the interactions between agents which happen within one region of the environment (mapped to an LP) will belong (for a period of simulated time) to the same simulation state. In this way, all the agents will observe a consistent simulation snapshot while simulation events access/modify their data structures.

In this way, if the event which created a new agent is rolled back, the policies taken to restore a previous consistent simulation state will release the memory associated with that agent too. Given that agents are identified by a UID, if the UID-generating module applies a rule similar to Equation (1), there is no need to rollback the state of that generator—simply, the UID associated with the rolled back agent will not be associated with any agent in the corrected simulation trajectory.

`RemoveAgent(agent_t agent).` Similarly to the creation of an agent, the simulation model might determine at a certain point in simulation that an agent no longer has to be part of the simulation. The `RemoveAgent()` API serves this purpose. This API function accepts an `agent_t` generated by a previous call to `AddAgent()`. The semantic associated with this API is such that, from that point in simulation on, any attempt to interact by API calls with the agent identified by the UID stored in `agent` should fail.

Regarding the release of the memory buffers used to keep track of that agent, we note that the event issuing a call to `RemoveAgent()` can be also rolled back. Therefore, the invocation to this API function is subject to rollbacks as well. With respect to the buffers used to keep the agent data structure, the same considerations made for the `AddAgent()` API function apply. On the other hand, the situation is different with respect to the UID kept in `agent_t`. Indeed, any subsequent interaction with that agent (even in the same event) should fail. Yet, if the call to `RemoveAgent()` is rolled back, that UID must become valid again.

As mentioned, we suggest organizing all UIDs in a fast hash table. The hash table can be augmented with a per-agent flag, telling

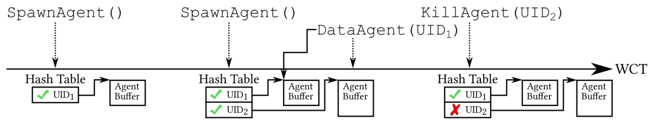

Figure 1: Agents management timeline.

whether some UID has been retracted from the simulation or not. A call to `RemoveAgent()` will therefore simply flag an agent as retracted. The UIDs which are retracted during the execution of the event can be stored in an ad-hoc buffer (e.g., a resizable array) kept in the data structure describing the event. Upon a rollback operation, the message buffer of the undone event is inspected, and all agents which were retracted can be “reintroduced” in the system by simply clearing their retraction flag. An agent is definitely (and consistently) removed from the simulation upon the GVT computation. In particular, when a committed event is removed from the LP queue (as in traditional Time Warp simulations), its retracted agents’ buffer can be inspected so as to remove from the hash map the associated UIDs, and free memory buffers. A parallel execution of the fossil collection phase after the GVT reduction is desirable, especially if it involves all the threads of the parallel runtime environment. Such a solution has a twofold benefit: i) it allows to reduce the overall time required to garbage-collect older buffers—we will describe other actions which shall be done after the GVT reduction—and ii) is introduces a “similar” delay on all worker threads, thus likely not generating a skew in the LVT of the various LPs which could possibly increase the rollback probability.

`GetAgentData(agent_t agent)`. Since the state of an agent, composed of its explanatory variables, can be freely organized by the simulation model developer, we have devised a simple API to manage agents’ states. This API function, given the UID of one agent, returns a pointer to a memory buffer which can be used to store the agent’s state. The size of this memory buffer should be at least of `user_data_size` bytes, i.e. the amount specified when creating an agent via `AddAgent()`. This memory buffer must be handled by the runtime environment from its allocation, via a call to `AddAgent()`, to its dismissal, after the computation of the GVT.

An overall timeline of the lifetime of an agent in one region (LP) of the simulation is depicted in Figure 1.

3.1.2 Supporting Agent Interactions and Decisions. We now move to describing the part of the API which allows multiple agents to interact when they are in the same portion of the environment (i.e., the same LP), or in proximity.

`CountAgents(void)`. As mentioned, ABMS can be particularly useful to study emergent behaviors. One fundamental aspect is to know the number of agents which are close to each other. Since in our system model each LP manages a portion of the environment, we can easily retrieve the number of agents which are currently in the LP running a simulation event by calling this API function.

The implementation of this API function should be quite simple, provided that there is a fast per-LP hash table, telling what are the UIDs of the agents currently registered at one LP.

`IterAgents(agent_t *agent_p)`. In order to support the simulation model developer at easily scanning through all the agents which are registered at a certain LP, this API function allows to implement an iterator. In particular, by repeatedly calling this function, the model will find stored in `agent_p` the id of the “next” agent registered in the region. Once the UID of the “next” agent is retrieved, the simulation model can interact with it in the desired way, by relying on any other API function. There is no strict requirement on the order according to which agents’ UIDs are returned to the simulation model developer, therefore the most efficient implementation can be picked for the runtime environment—this strongly depends on the way agents are registered within one region: if a per-LP hash table is used, then the “most efficient order” could be that of scanning through the table.

`RegisterNeighborInfo(void *neighbor_data)`. More complex decisions can be taken by the agents if they know about the state of the environment in proximity, not only of the portion of the environment in which they currently reside. While an agent-based model runs in a sequential or time-stepped environment can immediately access this information, speculative PDES poses an additional challenge. Indeed, in our system model, different regions are mapped to different LPs, which might have reached different simulation time instants due to the speculative nature of the simulation. Supporting this decision-making capability can be burdensome for the system. Consider, as an example, an agent registered in a region mapped to LP_x which has to decide whether to reach either LP_y or LP_z depending on the “crowdedness” of the destination regions. To make an informed decision, the agent has first to collect the number of agents currently residing at LP_y and LP_z. To implement this logic in traditional PDES, the model developer has to split this logic into multiple events to acquire the agent count in both LPs and then determine what is the correct destination. Given the speculative nature of the simulation, LP_y and/or LP_z might be forced to rollback. Additionally, since these “support events” could likely be *simultaneous events*, this might place an additional burden to the runtime environment, due to the need for some sort of *tie-breaking* function.

`RegisterNeighborInfo()` tries to solve these problems. The idea behind this API function is to implement a sort of publish/subscribe protocol between connected LPs. The model developer, at simulation startup, can define a portion of its simulation state which should be broadcast to all neighbor LPs, according to the current topology of the simulation—we will describe how the topology is supposed to evolve in section 3.2. In this way, the runtime environment is requested to make a local copy of the data of interest across neighboring LPs. This copy, which can be broadcast via traditional message passing, is superseded every time that a new version of the data is installed. Therefore, any time that an event needs to access data from a neighbor, a local copy can be inspected.

Two points deserve a discussion about an efficient implementation of this API function. First, not all events might update the portion of the state watched by `RegisterNeighborInfo()`. To reduce the number of messages which are silently exchanged, the runtime environment must detect whether the watched memory area has been updated by an event or not. To this end, depending on the actual organization of the runtime environment, we envisage two possible baseline implementations. One implementation, which is the one used in the experimental assessment presented in this paper, relies on a fast hash function to compute a hash of this memory buffer[4] to detect whether the last-executed event modified the buffer. An additional option, which is more costly, is to rely on memory protection mechanisms provided by, e.g., `mprotect()`.

The second point is associated with consistency of the values observed by the neighbors. Since updates must be transparently sent to neighbor LPs upon a state update, if the original event which caused the state transition is rolled back, also the updates towards the neighbor LPs should be annihilated. This is especially important because if an event executed at a neighbor LP read the retired value, also its execution must be rolled back, according to the global synchronization mechanism proper of Time Warp. To this end, we suggest relying on *control messages*, namely messages which are exchanged at the level of the runtime environment, destined to a certain LP, that are included in the event queue but are never delivered to simulation model handlers. These control messages allow to record that at a certain point in the simulation trajectory some relevant event happened, which must be subject to rollback operations, or which can be subject to the delivery of antimessages. The logic associated with these events is exactly that of updating the local copy of remote portions of the simulation state, once the timestamp associated with them is reached in the simulation. Annihilating such control messages via anti events cancels updates which have not yet been processed, while relying on the rollback operation allows to restore previous consistent snapshots in case that remote state updates are retracted. Of course, the buffers to store these copies must be rollbackable, so the same considerations for the buffers to keep agents explanatory variables apply.

`GetRegionData(direction_t direction, unsigned int *region_id, void **data_p)`. This API function is a simple entry point to the transparent local copies made by the runtime environment thanks to the `RegisterNeighborInfo()` API function described above. The goal of this API is to retrieve the data of a neighboring LP, which is made accessible by the runtime environment via the `data_p` pointer. To select what is the region which the user is interested in, `the direction_t` data type can be used. This allows to "navigate" the selected current environment topology — this concept will become clearer once we discuss how the topology is supported. In `region_id`, the runtime environment stores the id of the LP currently handling the portion of the environment which we target with `direction`.

3.1.3 Moving Agents Around. We now discuss the portion of the API which supports agents' mobility. We stress that the specification is general, and by no means we are bound to a physical embedding or a physical mobility model. To fully understand the functions which we have devised in our API, we shall explain why usually time-stepped simulation is considered easy-to-use in agent-based simulations. In many classical agent-based models from the literature (e.g., flocks [39], or game of life [18]), updates in the agents and/or the environment happen "globally". This means that it is not possible to make a definitive decision observing a subregion of the model. This approach, although simple, significantly clashes with the speculative PDES data partition-oriented approach, which is one of the reasons why the Time Warp synchronization protocol is so effective. To bridge the gap between these two approaches, agents' mobility decisions are made observing local information only and can be changed/retracted at any time by the model. This means that, once the simulation model logic decides that an agent should move from one region to another, this is the "best educated guess" that the model can make at that point of the simulation. This guess can be proven wrong later on by the model itself, and therefore the runtime engine should be informed of a modified decision—this also means that the mobility decision can be retracted completely.

To support this programming model without placing a high burden on the developer, we adopt a twofold strategy. On the one hand, we introduce the concept of *visits*: an agent has a set of regions to be visited, which can be modified at any time. These create a *visit list* which allows to describe the "ordering of places" in the topology which the agent will reach during the lifetime of the simulation[5]—we emphasize again that the topology can be "virtual", meaning that the visit list can be also used to construct at runtime a set of pending logical tasks for the agent or a combination of physical and logical actions. Every time that the agent makes a decision, it can modify this list arbitrarily. Two different agents have always a different visit list, but of course two agents can reach the same regions in the same order, depending on the logic that the simulation model is implementing. This is a versatile way to describe the way an agent explores and interacts in the environment, which can be easily adapted to many real-world problems. If agents do not have a large lookahead, the visit list simply boils down to the next region to visit.

`EnqueueVisit(agent_t agent, unsigned region, unsigned event_type)`. This API function allows to store at the end of the current visit list a new region to be reached. Once this visit is fired—the triggering API is `ScheduleNewLeaveEvent()` described below—the destination LP identified by `region` is hit by the `event_type` event.

`CountVisits(const agent_t agent)`. This API function allows to return the number of regions which are stored in the visit list which can be reached by the agent in its future moves. Past visits, namely regions which have already been visited, are not accounted for in this number. The `CountPastVisits(const agent_t agent)` API function can be used to this end.

`GetVisit(const agent_t agent, unsigned *region_p, unsigned *event_type_p, unsigned i)`. During the lifetime of the model, it might be of interest to inspect the future visit list. To this end, with respect to the `i`-th future visit, a call to this API will store into `region_p` the id of the LP to which the visit is associated, and into `event_type_p` the type of the event which will be scheduled when this visit is fired. If a user is interested in getting information about past visits, the `GetPastVisit(const agent_t agent, unsigned *region_p, unsigned *event_type_p, simtime_t *time_p, unsigned i)` API function can be used.

[4]We suggest using xxHash [9], an extremely-fast general-purpose hashing function working closely to the speed of RAM, and faster than `memcpy()`. Implementations for multiple programming languages are available.

[5]An example use-case scenario for visit lists is a delivery model: each agent has a set of items to be delivered to specific places in the environment which is defined at simulation startup. This list can dynamically change, e.g. due to traffic conditions.

With respect to the runtime environment implementation, we emphasize that the `GetPastVisit()` API function can produce a non-negligible memory footprint. Indeed, if the agents have a high mobility, especially in very long simulations, this past visit list might become very long. Since the semantic of this API is to allow to retrieve any past visit, also those associated with the committed portion of the simulation trajectory, we are not allowed to prune the past visit list even when a new GVT value is computed. Due to this consideration, we leave to the implementation the choice to always maintain the list of past visits, or start keeping them once the first invocation to `GetPastVisit()` is issued—in this latter case, of course, the model might not be able to observe the whole past visit list.

The last API functions to manage visit lists should allow to modify the current list. `AddVisit(agent_t agent, unsigned region, unsigned event_type, unsigned i)` allows to insert a new visit just before the `i`-th element in the future visit list, while `SetVisit(const agent_t agent, unsigned region, unsigned event_type, unsigned i)` allows to modify an already inserted visit (e.g., to modify the event associated with the firing of that visit). If a visit is to be removed from the future list, `RemoveVisit(agent_t agent, unsigned i)` can be used.

In Figure 2 we provide a graphical representation of the visit list and the management operations supported by our API. From an implementation point of view, the visit list can be realized either by relying on resizable arrays, or by using actual linked lists. While the latter implementation is easier, we encourage to pick the former, because it can provide a significant performance improvement thanks to memory locality. Moreover, such a compact data structure might be easily migrated around, e.g. across different NUMA nodes, thus being highly manageable by environments which enforce memory-oriented optimizations at runtime.

`ScheduleNewLeaveEvent(simtime_t time, unsigned int event_type, agent_t agent).` A visit list by itself does not identify at what simulation time the agent will reach the next region. To this end, the `ScheduleNewLeaveEvent()` API function allows to tell the runtime environment what is the current guess for the simulation time at which the "next hop" in the visit list will be taken. The simulation model is allowed to call this API function multiple times, therefore allowing to override the previous guess, provided that this call is issued by an event executed by the LP which is currently hosting the agent and before the visit event is fired—conversely, it will fail. If this call fails, it should not produce any runtime error, as the model developer might want to rely on this simple semantic (namely, "try" to make an agent leave) to keep the implementation simpler. If an agent is removed from the simulation by means of a call to `RemoveAgent()`, the firing of the leave event will never take place.

The only constraint to enforce is that `time` is in the future of the current simulation time at which the call takes place—this follows the traditional Time Warp synchronization protocol. To support this API function, we suggest the runtime environment to rely once again on *control messages*. These should be placed in the event queue of the LP which has called the `ScheduleNewLeaveEvent()` so that when its clock reaches the point in the simulation execution identified by `time`, then an event is sent to the LP associated with

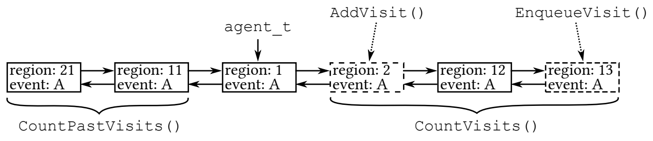

Figure 2: Visit management with respect to current region.

the next region to be visited. The payload of this event is the data structure (also with the user-defined payload) associated with the agent whose UID is `agent`.

This leave event can be rolled back as any other event. For the sake of efficiency, we suggest not to remove the data structure associated with `agent` from the sender LP, but rather to use the retraction flag which we have discussed before. These buffers will be released upon the computation of the GVT. While this might overall consume more memory, especially if an agent moves a lot, it can significantly reduce the overhead to support rollback operations. We envisage memory recollections policies, similar in spirit to the *cancelback protocol* [24], such that if free available memory falls below a certain threshold, buffers keeping older incarnations of agents which have left an LP can be released.

As a last note, we have to reconcile the just discussed optimization with the possibility that an agent visits the same region multiple times. In this case, it could be that in the speculative portion of the simulation trajectory, the same UID entry in the hash table should be associated with multiple incarnations of the same agent. In this case, it is sufficient to extend the hash table with a stack. In this stack, the agents are placed in descending timestamp order, thus allowing simulation events to always find on the top the newest incarnation of the agent. If a rollback operation occurs, then the no-longer consistent nodes—these are the agents' incarnations which have been rolled back—can be popped and freed. In this way, again, the top of the stack will point to the last consistent snapshot of the agent representation. In Table 1 we summarize the API which we have described so far.

3.2 Describing the Topology

As mentioned, the topology definition in an ABMS-oriented API must guarantee two fundamental aspects, along with extreme simplicity: i) high expressiveness, because the ABMS paradigm allows to represent very different aspects of the world; ii) the possibility to arbitrarily change the topology at runtime. As for the second point, this is fundamental because physical environments can be modified by the agents, or can be subject to changes related to the evolution of the agents (e.g., in disaster or rescue scenarios). Supporting a changing topology in a speculative runtime environment is a challenging aspect.

An initial topology must be specified at simulation startup, possibly along with details about the topology's custom configuration. This means model developers can specify shape of cells, how many sides and corners they share with other cells and how many cells they can consider as neighborhoods (i.e. the radius of the neighborhood). Despite this aspect is well studied in the literature, it allows model developers to generate arbitrarily complex topologies addressing their needs. To accomplish this, we comply with traditional file-based startup configurations—in our implementation, we rely on a JSON file. For the lack of space, we cannot provide the full JSON specification which we have envisaged. Several reference configuration files are anyhow available in the online repository.

The following is the set of topologies which we consider fundamental for ABMS—we also provide a graphical representation in Figure 3—which should be supported by the runtime environment, by means of the proposed API:

- *Square*: Each LP models a square cell. The overall environment can be either a square or a rectangle (this poses a limitation on the number of LPs that can be used in a simulation run). Four or eight neighbors can be reached from each cell, except from the boundaries.
- *Hexagon*: Each LP models a hexagonal cell. This topology is similar in spirit with the Square topology, but 6 neighboring cells can be reached from every cell not lying on the border.
- *Ring*: LPs are organized in a mono-directional ring. Agents can move from one LP to only one adjacent LP, in a single direction.
- *Bid-Ring*: A bidirectional ring is similar in spirit to a Ring topology, except for the fact that agents can move in two directions.
- *Torus*: A torus is similar in spirit to a Bid-Ring topology, except for the fact that this is a 3D topology in which from each LP there are 4 or 8 possible neighbors to reach.
- *Star*: A single LP is connected to all LPs. If an agent wants to move to any other LP, it has to pass through the central LP.
- *Fully Connected Mesh*: Every LP is connected to any other LP. This topology can create the highest burden on the runtime environment, due to the high level of communication that can exist across the network.
- *Graph*: This is a generic weighted and directed graph. Each LP is connected to a subset of all the LPs, and each connection (an edge) is associated either with a weight or with a probability. The agents can stochastically move around, taking into accounts the weights or the probabilities on each edge. In this case, for each node, the startup configuration file should provide a list of reachable nodes, with the associated weight/probability.

The latter topology allows to model changing environments at runtime. In particular, the graph topology can be implemented as a weighted fully-connected mesh, in which forbidden connections have an infinite weight or a zero probability[6]. In this way, by relying on the topology API, it is possible to change at runtime these weights/probabilities, in order to modify the connections across the environmental regions. Changes in the topology must be again rollbackable. Therefore, the runtime environment must associate the event which altered the topology with the new incarnation of the topology, and in case of a rollback, the change must be undone. This is not a trivial aspect, given that other events scheduled at other LPs might depend on a no-longer consistent topology. These events must be also rolled back, thus possibly generating cascading rollbacks. Moreover, changes in the topology might target LPs hosted at a remote kernel instance in a distributed environment, which must be notified of these changes via message passing. Again, we suggest splitting the variables describing the overall topology at each LP in a scattered way, relying on control messages to mark updates in it. Provided that these control messages are incorporated in the event queue of each LP, traditional rollback mechanisms allow to keep the overall topology consistent.

[6]Of course, if the graph is sparse, more compact data structures can be used. We leave the choice to the implementation of the runtime environment.

API function	Description
`AddAgent`	Create a new agent
`RemoveAgent`	Delete an agent
`GetAgentData`	Retrieve an agent's information
`CountAgents`	Retrieve the number of agents in the neighborhood
`IterAgents`	Return an iterator on the agents in the current region
`RegisterNeighborInfo`	Subscribe to a region information
`GetRegionData`	Retrieve information of a specific region
`EnqueueVisit`	Add a region to an agent's visit list
`CountVisits`	Return the number of entries in the pending visit list of an agent
`GetVisit`	Get an entry from an agent's visit list
`ScheduleNewLeaveEvent`	Initiate the movement of an agent towards the next entry in its visit list

Table 1: A summary of the agents API

To let the model developer alter the environment and fruitfully navigate it, we have devised the following API functions which concern with topologies. `NeighborsCount(unsigned int region_id)` and the `RegionsCount(void)` allow to retrieve from the current organization of the topology, respectively, the number of regions which can be reached from the one which is executing the event performing the call, and the total number of active cells. This is useful, e.g., to determine the reachability degree of a region, or to implement actions to observe and sense the surrounding environment. `NeighborsCount()` is especially useful on boundary regions, which might not be able to let agents navigate in all directions provided for the current topology (e.g., in a hexagon topology, central regions can move in six directions, but edge cells have fewer legal moves).

In case of a graph-based topology, the weights or the probabilities can be modified by relying on the `SetValueTopology(unsigned int from, unsigned int to, double value)` API function. This function updates the `value` associated with the edge between the `from` and `to` LPs. We emphasize again that both these LPs can be different from the one running the event causing the invocation of this function. Therefore, control messages should be used to enforce a causally consistent update at the `from` LP. The counterpart API function is `GetValueTopology(unsigned int from, unsigned int to)`, which allows to query the current value for the topology.

To allow the model developer to simply code the movements of the agent, it is fundamental to offer API functions which allow to cleverly query the current topology. In case of traditional grid-based topologies (i.e., different from graph-based), the concept of *direction* is intrinsic: in a grid-like environment the agent has a limited set of possible moves. For example, a square grid can lead to up to 8 directions, as in the Moore neighborhood topology. `GetReceiver(unsigned int from, direction_t direction)` returns the id of the neighbor which would be reached by moving

API function	Description
`NeighborsCount`	Return the number of currently reachable regions
`RegionsCount`	Return the total number of regions
`SetValueTopology`	Update the value associated with an edge
`GetValueTopology`	Return the value associated with an edge
`FindReceiver`	Pick a random neighbor region
`FindReceiverToward`	Return the next region to visit in the path to the destination
`ComputeMinTour`	Return the shortest path from source to destination as an array of LPs to visit

Table 2: A summary of the topology API

`direction`-bound. As an example, in the case of a hexagon grid, valid values for `direction` are *north*, *north-east*, *south-east*, *south*, *south-west*, and *north-west*.

At the same time, since ABMS is commonly stochastic, we also envisage the `FindReceiver(void)` API function which, depending on the current topology, picks a random neighbor. In grid-based topologies, the probability distribution should be considered uniform. For graph-based topologies, the distribution should account for the probabilities or weights associated with the edges.

The last two topology API functions deal with a more long-termed planning. `FindReceiverToward(unsigned int to)` returns the id of the next LP to visit in order to reach the LP identified by `to`. This API should consider the minimum-cost path, either considering the number of moves, or the weights in case of a graph-based topology. In our reference implementation we have relied on Dijikstra's pathfinding algorithm [14]. Any time that we run the pathfinding algorithm, we cache the computed Minimum Spanning Tree. This becomes beneficial also to speedup pathfinding to other destination LPs. Results are cached in per-LP data structures, and are flushed upon a change in the topology. To extract the complete tour with all visits, the modeler can rely on `ComputeMinTour(unsigned int source, unsigned int dest, unsigned int result[RegionsCount()])`, which returns an array keeping a list of LPs to visit. In this latter case, the model developer must take into account the fact that this list might change in the future of the simulation, because other concurrent LPs might alter the topology, as we have discussed. Nevertheless, this API function (used in conjunction with the concept of visit lists which we have described before) allows to prepare a tentative schedule for the actions of an agent in the immediate future—it can be modified at any time with the API which we have described before. In table 2 we summarize the API related to topology.

4 ASSESSMENT

4.1 Test-Bed Models and Environment

We have implemented the API and its associated runtime support in the ROOT-Sim runtime environment [35]. ROOT-Sim, developed in C11/Posix, supports a transparent deploy on distributed clusters of massively-parallel compute nodes.

For our assessment, we consider 3 synthetic models, and 3 real-world applications. The considered models are the following:

(1) *Stupid Model* [38]: in a toroidal region, several bugs (the agents) move randomly. Each cell produces food, at a certain production rate, which is eaten by the bugs which enter it—while eating, a bug grows in size. Bugs can move to another cell only if the destination cell is empty, and they select the empty neighboring cell with the highest food amount. With a certain probability, when a bug enters a cell, it either dies or reproduces. When a bug reproduces, it spawns five new bugs in surrounding cells if they are empty. If not enough empty cells are available, the bug spawns a smaller number of children. The simulation halts when the largest bug reaches a certain size.
(2) *Segregation* [42]: this social model is used to show how segregation among people happens, even though people (the agents) do not mind being surrounded by other agents of a different race. An agent is modeled by its race, ethnicity, economic status, etc. Multiple populations occupy a certain environment with random positions. An agent can be satisfied or not with its current location, in the sense that an agent is surrounded by a certain percentage of agents that are like itself. In the negative case, the agent moves to a different (emtpy) location.
(3) *Sugarscape* [15]: in a grid environment, every cell contains different amounts of sugar. The agents move around by targeting the cell filled with the largest amount of sugar. Once they eat, they metabolize and leave pollution. In a way similar to the stupid model, they can die and reproduce. Additionally, the agents can trade or borrow sugar, generate immunity or transmit diseases.
(4) *Terrain-Covering Ant Robots* (TCAR) [25]: This is an agent-based model particularly interesting for the assessment of rescue scenarios. If some kind of accident occurs in a region which is either unknown by the rescuers or altered by the accident itself, the first action in order to actually rescue the victims is to explore the whole region. The terrain is modeled as an undirected graph, accounting also for obstacles, in which ant robots move in every direction. While moving, they leave *pheromones*, i.e. they implement a node-couting algorithm, where each cell is assigned a counter which gets incremented whenever any robot visits it. Each ant picks its next direction by selecting the neighboring cell with the smallest visit counter value.
(5) *Robot Explore* [16]: A group of robots is set out into an unknown space, with the goal of exploring it. They keep a representation of the explored world, so as to determine which is the closest unexplored area they can reach, and they compute the shortest path to reach it. While moving around, the robots gather measurements of the environment and store them in their state. During the exploration, some accidents can make a region non-traversable. The robots explore independently, until one coincidentally detects another robot in its proximity. Once two robots meet, they can exchange the information sensed from the environment and make collaborative exploration decisions.
(6) *Tuberculosis* [33]: This model allows to simulate the spread of tuberculosis infections. It has been effectively used to study this phenomenon in the city of Barcelona. Individuals

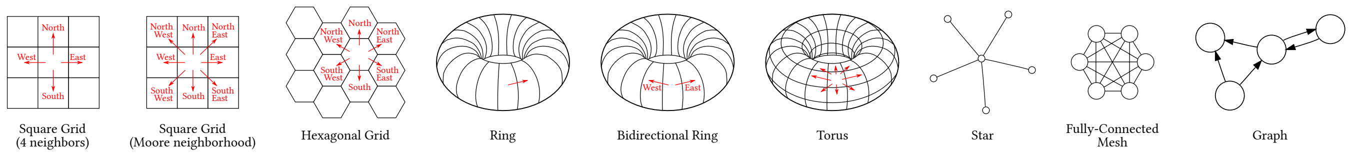

Figure 3: Supported topologies, and possible movements.

Model	Original Implementation	Using our API
Stupid Model	189	138
Segregation	83	110
Sugarscape	1072	152
TCAR	228 [7]	103
Robot Explore	500	332
Tuberculosis	1,115	654

Table 3: LOCs for the different Model Implementations

Model	# LPs	Involved Agents
Stupid Model	4,096	1 [8]
Segregation	10,000	5,000
TCAR	3600	48
Tuberculosis	1024	400,000

Table 4: Configurations of the models

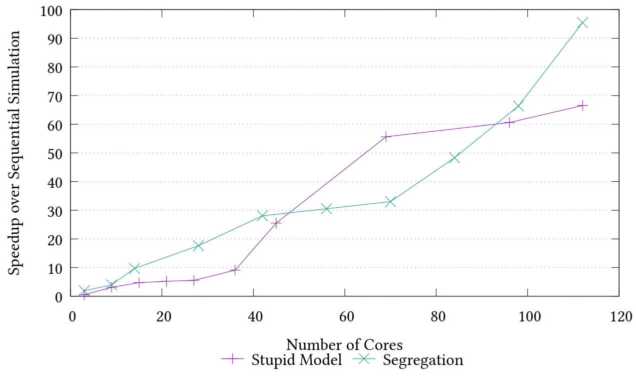

Figure 4: Results with Synthetic Models.

(the agents) can move around. They can be healthy, infected, sick, under treatment, or susceptible to relapse. Depending on their interactions, healthy individuals might become infected. Sick agents might be susceptible to death. This is an overall epidemiological model which requires a very large number of agents, with complex state transitions, to allow the identification of emergent behavior related to the spread of the disease.

4.2 Effects on Programmability

While assessing the effects on programmability is not easy, especially in contexts which involve models coming from multiple domains, we are taking into account here classical agent-based models from the literature, and we compare their classical implementations with ones made relying on the proposed API. For each of the aforementioned models, we have considered the original implementation (where available). We have compared the Lines of Code (LOCs) of the original implementation with the LOCs of our re-implementations using the discussed API.

Table 3 shows the comparison. As it can be seen, except for segregation model, the line count is always smaller when implementing the models by relying on the proposed API. It is interesting to note that this is the case also when dealing with very simple (toy) models, such as the Stupid Model. This is an indication of the fact that the API is effective at capturing the needs of ABMS, thus reducing the burden on the model developer which can benefit from the performance speedup and the increased amount of working memory traditionally offered by distributed PDES.

4.3 Performance Assessment

In the experimental assessment, we report performance data of 4 of the aforementioned models. We have used a cluster of three medium size heterogeneous servers connected via a 1 GB dedicated Ethernet connection. Two servers are 32-core (AMD Opteron 6168) machines equipped with 64 GB of RAM. One is a 48-core (AMD Opteron 6168) machine equipped with 128 GB of RAM. All machines run Linux 4.9.0. We have used gcc 8.2.0 to compile the software, and OpenMPI 3.1.3 as the MPI runtime environment. In the experiments, the total number of threads has been varied from 3 to 112. The MPI assignment policy has been set so as to ensure that all three machines are used, by distributing the threads on the cluster in a round-robin fashion. This is a worst-case scenario, especially when the number of threads is low, because we always incur in the communication cost even though a single machine could support message passing by relying on shared memory.

The models used and their configurations are reported in Table 4. To show the viability of our proposal from a performance point of view, we report the speedup of the distributed runs over a sequential simulation. Sequential simulations are extremely optimized, as they are based on a fast O(1) scheduler based on a calendar queue [5]. We remind that the experimental setup is a worst case, due to the fact that we always incur in increased network activity when we rely on a reduced number of threads, due to the distribution of the threads in the cluster.

[7] The original paper in [25] does not provide a reference implementation. Therefore, we report here the LOCs associated with a re-implementation of the model, according to the specification, which does not rely on our API.

[8] This is the initial agent count. During the simulation, the number of agents increases.

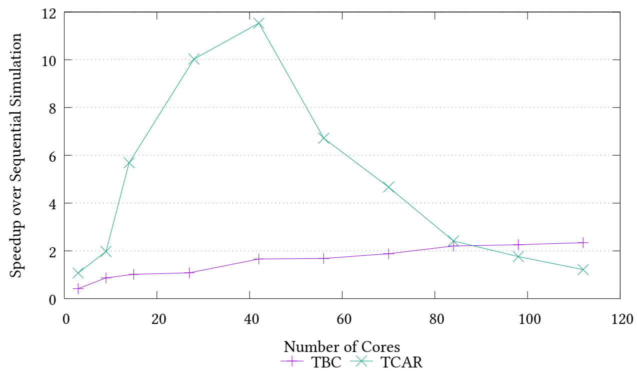

Figure 5: Results with Real-World Applications.

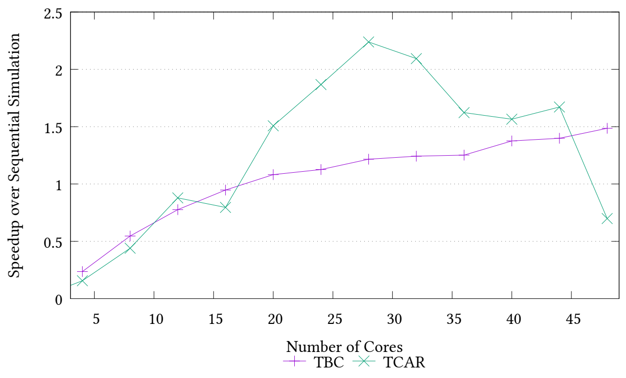

Figure 6: Results using a single node.

In Figure 4 we report the performance data related to the Stupid Model and Segregation synthetic benchmarks. It is interesting to note that both models exhibit a linear speedup, despite the fact that the event granularity is quite small (on the order of tens of microseconds) and the amount of events exchanged across the LPs is non-minimal—this latter point is related to the fact that in both models we rely on the `RegisterNeighborInfo()` API which transparently sends control messages to neighboring LPs upon any update, and the number of agents in the model (especially in the segregation case) is quite high. The reason behind this point is that the scale of the model can be significantly increased (up to 5,000 LPs in the Segregation model case) due to the fact that simulations states are very small, and so is the number of explanatory variables used to describe an agent. Anyhow, this result suggests that if these models are used (as they are intended to) as skeletons to build more complicated logic, then they will benefit from the increased computing power offered by distributed PDES.

In Figure 5 we report the performance results for two real-world simulation models, namely tuberculosis (TBC in the plot) and TCAR. With respect to TCAR, we can see that after a certain number of concurrent threads, the performance observes a significant drop. This is a foreseeable result, taking into account the configuration reported in Table 4. Indeed, there is a high imbalance between the number of active agents and the regions in the model. Given the distributed nature of the simulation, the probability that LPs observe a skew in the simulation time is high. This, in its turn, affects the overall efficiency of the simulation—in our runs, it got as low as 15% in some configurations. While this is a phenomenon which would have appeared independently of the API used to implement the model, we can note that the proposed API and its implementation is resilient to this unfavorable scenario up to a non-negligible number of distributed threads (42).

With respect to TBC, we note that this is the model with the highest degree of parallelism, due to the fact that one thread manages a reduced number of LPs (9 LPs per thread is the minimum). This is a scenario which is known to increase the rollback probability. Moreover, this is also exacerbated by the fact that the number of agents in the simulation, compared to the size of the environment, is quite large. The proposed API, in this case, is able to provide anyhow a speedup, although minimal. It is important to note that this is a clear case in which resorting to our proposal allows to make unfeasible problems feasible, due to the possibility to overcome the memory wall thanks to the increased number of nodes used for the simulation. In this specific model, the sequential simulation has shown a memory footprint of more than 16 GB, which could not be manageable when using off-the-shelf commodity hardware.

For the sake of completeness, in Figure 6 we report performance data taken when running on the largest single node of our cluster, for the real-world applications. Here, we do not pay communication costs, but the trends of the curves are perfectly similar to the ones shown in Figure 5, thus showing that the proposed API and its reference implementation is also resilient to network delays.

5 CONCLUSIONS AND FUTURE WORK

In this paper we have introduced an API specification for ABMS in Time Warp runtime environments. This API has been shown to be effective at implementing very different agent-based models in a compact and expressive way. An implementation of our API has also shown that it is possible to obtain non-minimal speedup also in very simple (toy) models. We therefore consider this as an important step ahead at disclosing the power of PDES to domain experts which should not be exposed to the complexity of speculative synchronization.

ACKNOWLEDGMENTS

We gratefully thank Philipp J. Andelfinger for the amazing job and the invaluable feedback provided to us while carrying out the reproducibility of our artifacts.

REFERENCES

[1] Sameera Abar, Georgios K. Theodoropoulos, Pierre Lemarinier, and Gregory M.P. O'Hare. 2017. Agent Based Modelling and Simulation tools: A review of the state-of-art software. *Computer Science Review* 24 (may 2017), 13–33. https://doi.org/10.1016/j.cosrev.2017.03.001

[2] Theodore T. Allen. 2011. *Introduction to Discrete Event Simulation and Agent-based Modeling.* Springer London, London. https://doi.org/10.1007/978-0-85729-139-4

[3] E. Bonabeau. 2002. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences* (2002). https://doi.org/10.1073/pnas.082080899 arXiv:1709.03423

[4] Andrei Borshchev and Nikolay Churkov. 2018. Anylogic cloud: cloud-based simulation analytics. In *Proceedings of the 2018 Winter Simulation Conference (WSC).* IEEE Press, 4245–4245.

[5] Randy Brown. 1988. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Commun. ACM* 31, 10 (1988), 1220–1227.

[6] Christopher D Carothers, Kalyan S Perumalla, and Richard M Fujimoto. 1999. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation* 9, 3 (1999), 224–253.

[7] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2015. RAMSES: Reversibility-based agent modeling and simulation environment with speculation support. In *Proceedings of Euro-Par 2015: Parallel Processing Workshops*, Sasha Hunold, Alexandru Costan, Domingo Ginenéz, Alexandru Iosup, Laura Ricci, María Engracia Gómez Requena, Vittorio Scarano, Ana Lucia Varbanescu, Stephen L. Scott, Stefan Lankes, Josef Weidendorfer, and Michael Alexander (Eds.). LNCS, Springer-Verlag, 466–478. https://doi.org/10.1007/978-3-319-27308-2_38

[8] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2017. Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. *ACM Transactions on Modeling and Computer Simulation* 27, 2 (may 2017), 1–26. https://doi.org/10.1145/3077583

[9] Yann Collet. 2015. xxHash: Extremely Fast Hash Algorithm. http://www.xxhash.com/

[10] Nicholson Collier and Michael North. 2013. Parallel agent-based simulation with Repast for High Performance Computing. *SIMULATION* 89, 10 (oct 2013), 1215–1235. https://doi.org/10.1177/0037549712462620

[11] Andrew M. Colman. 1998. The complexity of cooperation: Agent-based models of competition and collaboration. *Complexity* (1998).

[12] Gennaro Cordasco, Rosario De Chiara, Ada Mancuso, Dario Mazzeo, Vittorio Scarano, and Carmine Spagnuolo. 2012. A framework for distributing agent-based simulations. In *Proceedings of Euro-Par 2011: Parallel Processing Workshops*, Michael Alexander, Pasqua D'Ambra, Adam Belloum, George Bosilca, Mario Cannataro, Marco Danelutto, Beniamino Martino, Michael Gerndt, Emmanuel Jeannot, Raymond Namyst, Jean Roman, Stephen L. Scott, Jesper Larsson Traff, Geoffroy Vallée, and Josef Weidendorfer (Eds.). Springer Berlin Heidelberg, 460–470. https://doi.org/10.1007/978-3-642-29737-3_51

[13] Biagio Cosenza, Nikita Popov, Ben Juurlink, Paul Richmond, Mozhgan Kabiri Chimeh, Carmine Spagnuolo, Gennaro Cordasco, and Vittorio Scarano. 2018. OpenABL: A Domain-Specific Language for Parallel and Distributed Agent-Based Simulations. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-319-96983-1_36

[14] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (dec 1959), 269–271. https://doi.org/10.1007/BF01386390

[15] Joshua Epstein and Robert L. Axtell. 1997. *Growing Artificial Societies: Social Science from the Bottom Up.* MIT Press. https://doi.org/10.1007/s13398-014-0173-7.2

[16] Dieter Fox, Jonathan Ko, Kurt Konolige, Benson Limketkai, Dirk Schulz, and Benjamin Stewart. 2006. Distributed Multirobot Exploration and Mapping. *Proc. IEEE* 94, 7 (2006), 1325–1339. https://doi.org/10.1109/JPROC.2006.876927

[17] Richard M Fujimoto. 1990. Parallel Discrete Event Simulation. *Commun. ACM* 33, 10 (1990), 30–53.

[18] M Gardner. 1970. Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American* 223, 4 (1970), 120–123.

[19] Nigel Gilbert and Pietro Terna. 2000. How to build and use agent-based models in social science. *Mind & Society* 1, 1 (mar 2000), 57–72. https://doi.org/10.1007/BF02512229

[20] Jan Himmelspach and Adelinde Uhrmacher. 2009. The JAMES II framework for modeling and simulation. In *HiBi09 - 2009 International Workshop on High Performance Computational Systems Biology*. https://doi.org/10.1109/HiBi.2009.20

[21] Mike Holcombe, Simon Coakley, and Rod Smallwood. 2006. A general framework for agent-based modelling of complex systems. In *Proceedings of the 2006 European conference on complex systems.* European Complex Systems Society Paris, France.

[22] Hitoshi Iba. 2013. *Agent-Based Modeling and Simulation with Swarm.* Chapman and Hall/CRC. https://doi.org/10.1201/b15024

[23] David R Jefferson. 1985. Virtual Time. *ACM Transactions on Programming Languages and System* 7, 3 (1985), 404–425.

[24] David R Jefferson. 1990. Virtual time II: storage management in conservative and optimistic systems. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing (PODC '90).* ACM, New York, NY, USA, 75–89. https://doi.org/10.1145/93385.93403

[25] Sven Koenig and Yaxin Liu. 2001. Terrain Coverage with Ant Robots: a Simulation Study. In *Proceedings of the fifth international conference on Autonomous agents (AGENTS).* ACM, 600–607. https://doi.org/10.1145/375735.376463

[26] Justin M LaPre, Elsa J Gonsiorowski, and Christopher D Carothers. 2014. LORAIN: a step closer to the PDES 'holy grail'. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of Advanced Discrete Simulation (PADS).* ACM Press, New York, New York, USA, 3–14. https://doi.org/10.1145/2601381.2601397

[27] Michael Lees, Brian Logan, and Georgios Theodoropoulos. 2007. Distributed simulation of agent-based systems with HLA. *ACM Transactions on Modeling and Computer Simulation* (2007). https://doi.org/10.1145/1243991.1243992

[28] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. 2005. MASON: A multiagent simulation environment. *Simulation* 81, 7 (2005), 517–527. https://doi.org/10.1177/0037549705058073

[29] Shingo Mabu, Kotaro Hirasawa, and Jinglu Hu. 2007. A Graph-based Evolutionary Algorithm: Genetic Network Programming (GNP) and its Extension sing Reinforcement Learning. *Evolutionary computation* 15, 3 (sep 2007), 369–98. https://doi.org/10.1162/evco.2007.15.3.369

[30] Charls M Macal and Micheal J North. 2005. Tutorial on Agent-Based Modeling and Simulation. *Proceedings of 2005 Winter Simulation Conference* (2005).

[31] Sally a. McKee. 2004. Reflections on the memory wall. In *Proceedings of the first conference on Computing Frontiers.* ACM Press, New York, New York, USA, 162. https://doi.org/10.1145/977091.977115

[32] Steven Mithen. 2002. Stepping out: a computer simulation of hominid dispersal from Africa. *Journal of Human Evolution* 43, 4 (oct 2002), 433–462. https://doi.org/10.1016/S0047-2484(02)90584-1

[33] Cristina Montañola-Sales, Joan-Francesco Gilabert-Navarro, Josep Casanovas-Garcia, Clara Prats Soler, Daniel López Codina, Joaquim Ribas Valls, Pere Joan Cardona Iglesias, and Cristina Vilaplana. 2015. Modeling tuberculosis in Barcelona. A solution to speed-up agent-based simulations. In *Proceedings of the 2015 Winter Simulation Conference.* IEEE Computer Society, 1295—-1306.

[34] Cristina Montañola-Sales, Bhakti S.S. Onggo, Josep Casanovas-Garcia, Jose María Cela-Espín, and Adriana Kaplan-Marcusán. 2016. Approaching parallel computing to simulating population dynamics in demography. *Parallel Comput.* (2016). https://doi.org/10.1016/j.parco.2016.07.001

[35] Alessandro Pellegrini and Francesco Quaglia. 2014. The ROme OpTimistic Simulator: A tutorial. In *Proceedings of the Euro-Par 2013: Parallel Processing Workshops*, Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Constan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, Stephen L. Scott, and Josef Weidendorfer (Eds.). LNCS, Springer-Verlag, 501–512. https://doi.org/10.1007/978-3-642-54420-0_49

[36] Alessandro Pellegrini, Francesco Quaglia, Cristina Montanola-Sales, and Josep Casanovas-Garca. 2016. Programming agent-based demographic models with cross-state and message-exchange dependencies: A study with speculative PDES and automatic load-sharing. In *Proceedings of the 2016 Winter Simulation Conference (WSC).* IEEE, 955–966. https://doi.org/10.1109/WSC.2016.7822156

[37] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2009. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings - Workshop on Principles of Advanced and Distributed Simulation, PADS.* IEEE, 45–53. https://doi.org/10.1109/PADS.2009.24

[38] Steve F Railsback, Steve Lytinen, and Volker Grimm. 2005. *StupidModel and Extensions: A Template and Teaching Tool for Agent-based Modeling.* Technical Report. Swarm Development Group. 7 pages.

[39] Craig W. Reynolds. 1987. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics* 21, 4 (aug 1987), 25–34. https://doi.org/10.1145/37402.37406

[40] Paul Richmond and Daniela Romano. 2008. Agent based gpu, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the gpu. In *Proceedings International Workshop on Supervisualisation.*

[41] Xavier Rubio Campillo, Jose María Cela, and Francesc Xavier Hernàndez Cardona. 2012. Simulating archaeologists? Using agent-based modelling to improve battlefield excavations. *Journal of Archaeological Science* 39, 2 (feb 2012), 347–356. https://doi.org/10.1016/j.jas.2011.09.020

[42] Thomas C Schelling. 1978. Sorting and Mixing. In *Micromotives and Mmacrobehavior.* W W Norton & Co, Chapter 4, 270.

[43] Vinoth Suryanarayanan, Georgios Theodoropoulos, and Michael Lees. 2013. PDES-MAS: Distributed simulation of multi-agent systems. In *Procedia Computer Science.* https://doi.org/10.1016/j.procs.2013.05.231

[44] Seth Tisue and Uri Wilensky. 2004. Netlogo: A simple environment for modeling complexity. In *Proceedings of the International Conference on Complex Systems (ICCS).* NECSI, 1–10.

[45] Roberto Toccaceli and Francesco Quaglia. 2008. DyMeLoR: Dynamic Memory Logger and Restorer Library for Optimistic Simulation Objects with Generic Memory Layout. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation.* IEEE Computer Society, 163–172. https://doi.org/10.1109/PADS.2008.23

[46] George Vulov, Cong Hou, Richard Vuduc, Richard Fujimoto, Daniel Quinlan, and David Jefferson. 2011. The Backstroke framework for source level reverse computation applied to parallel discrete event simulation. In *Proceedings of the 2011 Winter Simulation Conference (WSC).* IEEE, 2960–2974. https://doi.org/10.1109/WSC.2011.6147998

[47] P Wittek and X Rubio-Campillo. 2012. Scalable agent-based modelling with cloud HPC resources for social simulations. In *Proceedings of the 4th International Conference on Cloud Computing Technology and Science (CloudCom).* IEEE Computer Society, 355–362. https://doi.org/10.1109/CloudCom.2012.6427498

[48] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. 2018. Exploring Execution Schemes for Agent-Based Traffic Simulation on Heterogeneous Hardware. In *Proceedings of the 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT).* IEEE Computer Society, 1–10. https://doi.org/10.1109/DISTRA.2018.8601016