Reconciling Simulation and Distributed Deployment in a MAS Development Framework

Andrei Olaru[0000−0002−2718−9195] and Ionuț Hodoroagă

Department of Computer Science and Engineering, National University of Science and Technology POLITEHNICA Bucharest, 313 Splaiul Independentei, 060042 Bucharest, Romania andrei.olaru@upb.ro, ionut.hodoroaga@stud.acs.upb.ro

Abstract. In a world where agents are attracting renewed attention as a model for distributing artificial intelligence applications, the ability to test and simulate systems of agents becomes more important. However, agent-based simulation platforms and network-distributed multi-agent system deployment frameworks traditionally differ strongly in their approach to agent interaction and execution model, although each could benefit from featuring the capabilities of the other.
Following in the tracks of other works that have attempted to bridge the gap for specific cases, this paper makes a primary investigation on the requirements that a framework would need to satisfy to support both large-scale simulations and real-life deployments with the same agent code.
We present a minimal set of principles upon which features can be developed, relating to interaction support, execution control, and environment modeling. We developed a set of features that allow compatibility between an ABM application and an existing MAS framework, and a proof-of-concept implementation of a classic ABM application using a framework originally built for network-based deployment.

Keywords: Multi-agent systems · Multi-agent frameworks · Agent-Based Modeling and Simulation.

1 Introduction

In the world of multi-agent systems (MAS) and the software platforms underpinning them, there is an important distinction between two types of applications, which are supported by different types of platforms.

On the one hand, there are MAS applications where agents interact via agent communication languages or other communication protocols, can execute on different machines, and have a complex lifecycle based on multiple behaviors and a high level of asynchronicity. These applications are underpinned by frameworks such as Jade and SPADE [6,23]. While these are generally referred to as "MAS frameworks", to better make the distinction and not create confusion, let us call these *network-based deployments*, since one of their central features is the support of communication over the network. These applications are deployed *in real life*, or *IRL*.

In *simulations* based on *agent-based models*, the number of agents is large, interactions can be complex, and it all happens on a single machine, or in a tightly coupled execution environment such as an HPC computing cluster. There are visualizations that can be used to monitor the progress of the simulation, and the user is able to control the speed of the simulation or pause it to investigate the details of a particular instantaneous state. These applications are *simulations*, and we will abbreviate this type of scenario as *Sim*.

ABM and MAS applications are not *fundamentally* different. In both scenarios, there are agents, which interact with the environment via perceptions and actions, and which interact among each other, via objects in the environment, via events, or via direct communication.

Many times, network deployments and agent-based simulations are seen as different fields, with no reason to interoperate the two. Indeed, some simulations, such as crowd simulations, are meant to simulate a process which happens in reality and which features non-software entities. However, simulation is also a means of *testing* the functionality of a system formed of a large number of software agents, or the interaction between software agents and entities that, in the real deployment, are non-software, e.g. humans. Simulations are how we can tweak the behavior of agents so that, when deployed in real life, they will behave correctly. This is especially useful in applications concerning mobile robots and robot swarms [17,9,18] and traffic control [31,2], in which the simulation precedes the deployment of agent behaviors into the real-life application. In traffic-related applications, for instance, once a complex behavior is implemented for agents – either agents controlling traffic lights, or agents controlling cars, or both – it is desirable that the same code is used, as developed, in real life.

Our vision is to have a framework for agent deployment that allows deploying agents in real life, but also allows *the same* agents to be tested in simulations, and by *the same* we mean agents with the same code and the same behavior – the same "*agent mind*". While this has been attempted before, both in a general purpose approach [10,27,8] and for specific application domains [1,31], currently mainstream ABM frameworks offer no possibility of deploying the simulated agents in real life, with the same code (e.g. deploy Netlogo agent code to control a robot), and deployment frameworks such as JADE or SPADE cannot support large-scale simulations.

We are in the course of defining the architecture of a framework that allows agents to be deployed in a real-life scenario, interacting with each other via the network and interacting with the environment, but also to be simulated in a controlled environment, with simulations being efficient and execution is controllable.

Network-based deployment of agents, such as in Jade or SPADE, means that agents need to have a lifecycle usually based on behaviors, sometimes based on cognitive models such as BDI, are able to interact asynchronously over a network, potentially using secure communications, and interact with their environment directly or, in the Agents and Artifacts (A&A) meta-model, via artifacts. But *IRL* applications would benefit from the possibility to simulate the agent behaviors

and interactions prior to deployment, in a controlled execution environment, but without the performance overhead generated by more capable interaction infrastructures, and with the possibility of controlling the pace of the simulation [10].

In an agent-based simulation, such as in NetLogo, Repast, or GAMA [32,11,29], agents interact mostly with their neighborhood, and implement a more reactive approach to internal agent modeling, either executing in a step-wise (tick-based) manner, or via reaction to events (reflexes). But *Sim* applications would benefit from the possibility of more advanced internal agent modeling and behavior (e.g. a BDI approach) and, for some applications, from the possibility of deploying the agents, as they are, in a real-world setup.

The question we are tackling is what would be the architectural principles around which to build such a framework, that allows both simulation and real-life, network-based deployment. We investigated the challenges that arise and studied several use cases in search of developing a *minimal* but *complete* set of requirements that would satisfy both scenarios. Upon these requirements, we have built an infrastructure of necessary functionality, focusing on modeling concerns but also keeping in mind the potential for performance in large-scale simulation.

To put the principles that we have developed to the test, before moving on to more complex applications, we have implemented a proof-of-concept implementation of a classic ABM scenario in Flash-mas, a modular, flexible agent deployment framework developed within our department [19,20]. While we have noticed that the Flash-mas framework is particularly appropriate for our objective, thanks to its modularity and focus on contexts, the goal of this research is to identify general principles that can be extended to other MAS frameworks. By integrating ABM features in Flash-mas, we can also help establish Flash-mas as a capable agent deployment framework suitable of a variety of scenarios.

The purpose of this research, of which this paper showcases only the first steps, is twofold. On the one hand, it identifies several principles that can be followed by frameworks that wish to gain the ability to handle both scenarios; on the other hand, it showcases the design of several components that enable the compatibility between *Sim* and *IRL* scenarios and shows how Flash-mas can use these components to become an agent deployment framework that is able to handle both agent-based simulation and network-distributed applications.

2 Related Work

There is a significant body of work in MAS frameworks, as well as in ABM simulations. However, there are relatively few works at the intersection of the two. We will review some of them here and highlight how they are related to this work.

2.1 From Simulation to Real-Life Deployment

While some simulations are meant to replicate real-life processes just to observe what is the result of environmental conditions, some simulations are used as a testbed for entity behavior before entities are deployed in the real world. These simulations are using application-specific simulation platforms.

The Aerialist is a test bench for drone control software, focused on simulating environment conditions and specific control issues [16]. Barone et al [4] introduce a simulation framework for military applications, using a derivative of NetLogo for the simulation, as well as a specialized simulator for realistic network communication conditions. In the field of networking, Pinyoanuntapong et al [24] present the effort of transferring behaviors tested in simulation to real-life, where the behaviors, policies and knowledge need to be transferred to specialized devices, such as network switches and routers. And Al-Zinati and Zalila-Wenkstern introduce MATISSE 2.0 as a testbed for agent-based transportation systems, before they are deployed in real-life

When transferring policies and behaviors from simulation to real-life deployments (what is called "sim to real"), there are several gaps that can be identified [9]: the simulation gap, derived from the forced synchronicity in simulated agent execution; the observation gap, derived from the reduced observation horizon in the real world; and the communication gap, resulting from imperfect real-world communication. These gaps can be, at least in part, covered by simulation components appropriately replicating realistic environments. However, one gap that occurs in many applications is the lack of software support that allows for a seamless sim-to-real transfer for that specific application domain.

There is a significant body of work dealing with the transfer of reinforcement learning policies from simulation to real-world environments [30], especially in application domains such as robotics [17] and autonomous vehicle control [9]. However, many of these transfers focus on how to transform input/output streams from simulated inputs and feedback loops to real-life interfaces. In these works, agent behavior is represented by a policy. In agent-oriented engineering, however, there is considerably more focus on the internal architecture of agents and on features offered by the framework that enable communication and the relationship with the environment. Internal agent architectures can rely on more than applying a learned policy, such as, for instance, behavior-based architectures or models inspired by the BDI approach [26].

2.2 Bringing Simulation Functionality to MAS Frameworks

To solve the issue of simulating agents with more advanced behaviors, there have been several attempts to bring simulation capabilities to existing MAS frameworks, especially Jade and JaCaMo, with the goal of validating BDI models through simulation [3].

Collier et al [12] introduces the concept of *Hybrid Simulation* – the ability to integrate different sub-simulations in the same scenario, each potentially differently modelled, with different techniques. The authors highlight the lack of tools for integration of ABM with other techniques [25].

Jagutis et al [15] create an architecture for bringing simulation features to the microservices-based ASTRA-MAMS agent framework, identifying two distinct microservices for simulation: one is the environment and the other is an agent-oriented service, running the behaviors of agents. The environment sends to the agents percepts and affordances, whereas agents send actions. In our work we also observe this separation, but we take it further towards a more controllable execution.

Palanca et al [22] illustrate the need for agents that combine different types of behaviors in the same architecture, such as reactive behaviors but also BDI-based decisional behaviors. The implementation is in SPADE 3 and the BDI behaviors are specified using AgentSpeak. We envision that by integrating simulation features in Flash-mas we will also enable a variety of hybrid agent systems.

Several solutions propose interoperation between agent deployment frameworks and ABM frameworks or an ABM-oriented approach, based on providing agents with perceptions and obtaining from agents actions to be executed in the environment. One example is the ABM-BDI approach [28]. However, there is no manner in which agents can access functionality from the environment in a more responsive and efficient manner, such as obtaining, on demand, information about their surroundings, or about other agents. Rather, the interface between environment and agents is heavy and unsuited for large-scale simulation. While this could be solved by offering all the possibly accessible information in the perception, this would greatly decrease the efficiency of the implementation.

Cardoso created a mechanism for interoperation between two popular frameworks – Jade and Repast, to enable simulation of Jade agents using the Repast framework [10]. The complexity and lack of modularity of Jade, however, hinder the process, as well as the fact that Jade allows blocking calls. The simulation requires modifying Jade itself, and does not use the exact same Jade agent code, but supports automated bidirectional conversion between agent code for normal Jade and for the simulated version.

The SimJade Architecture integrates Jade with the PeerSim high-performance simulator by intercepting messages and calls to the AMS and DF services [8], but there is no interaction between agents and the environment.

Several initiatives have attempted to interoperate Jason or JaCaMo [7] with simulation features, as JaCaMo is a popular, and powerful, agent-oriented programming framework with rich agent behavior specification derived from AgentSpeak. Some attempts have been made to interoperate JaCaMo with the SUMO traffic simulator [5,31], but they are limited to application-specific solutions and rely on Jason submitting explicit requests that are destined to reach the simulator.

JaCaMo-sim [27] attempts to integrate general-purpose simulation features into the platform. Like other attempts, it focused on uncoupling the agent behavior from the actual reading of perceptions, receipt of events, and execution of actions, such that the environment could be simulated and the pace of the execution controlled. However, the work has unfortunately not continued.

Grosjean et al [14] also work in improving the model of ABM frameworks, by decoupling the application (the 'thematic' model) from communication and distribution mechanisms (the 'distribution' model), but remaining in the area of HPC simulations and not crossing into more loosely coupled network deployments.

SARL, a general-purpose agent-oriented language can run on different execution platforms [13], such as Janus, appears suited for both ABM scenarios and network-based deployments, however it remains to be seen if it can flexibly integrate a wider range of environment contexts.

Baiardi et al introduced the ability to simulate JaKtA agents using Alchemist – a Discrete Event Simulator. The authors make a thorough analysis of requirements and use a UAV scenario, but the simulation benefits from the fact that agents are implemented in a domain-specific language, rather than in a general-purpose one.

3 Approach, Requirements, and Principles

Our research deals with a design that enables the same agent code to be deployed in both *Sim* – agent-based simulation – and in *IRL* – real-life scenarios.

A first question here is "*what is agent code and is there anything else in an agent*?". We see the *agent code* as equivalent to the *agent mind*, as opposed to the *agent body*. The *agent mind* represents its behavior, its decision algorithm, internal state and knowledge management, input processing, etc. Conversely, the *agent body* contains all elements that are associated with the agent but are not *internal* to its decision process, such as the actual sensors and actuators, for a robotic embodiment; extrinsic properties such as its position in space; or any other component that mediates between the agent and the environment. Examples of *agent code* are shown, in different languages, in Figure 3.

A second question is "*what are the connections that the agent mind needs to other components*?". We are interested in all connections that need to behave differently in the *Sim* and the *IRL* scenarios.

And the last question is "*what is the minimal set of requirements that need to be satisfied by a multi-agent deployment framework such that the same agent code is compatible with both* Sim *and* IRL *execution*?"

3.1 Sketching the Requirements

Any agent deployment framework must ensure that agents are *situated in* and can *interact with* the environment, and that they can interact with each other via *communication protocols*.

In relation to the environment, the agent must be able to *observe* and *perceive* at least a part of the environment, and to *execute actions* that affect the environment or affect the situation of the agent body in the environment (e.g. agent movement). In *Sim*, when an agent performs a `move(position)`, the simulation environment changes the extrinsic *position* property of the agent, so that

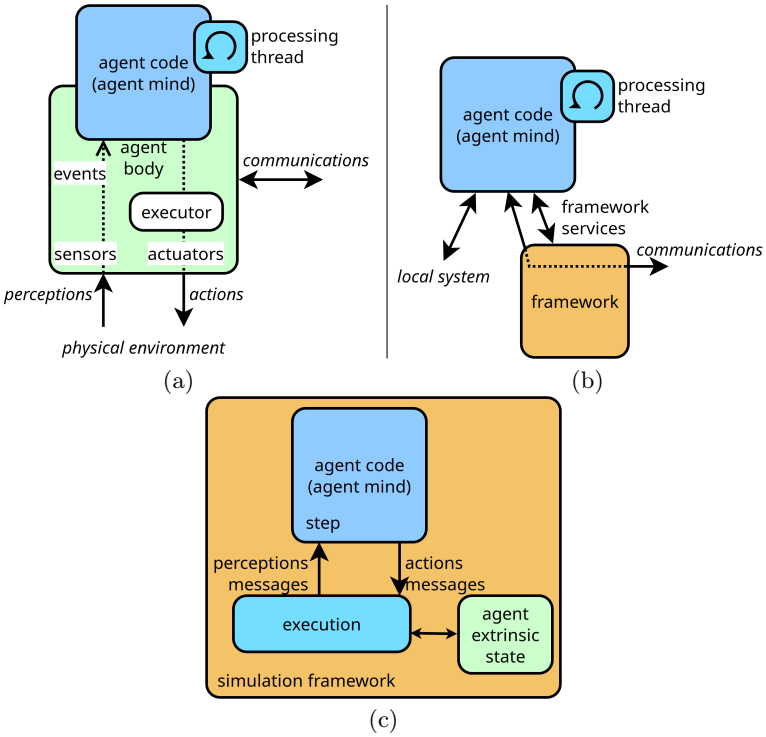


Fig. 1. The relation of the agent code (the agent 'mind') with other components of the scenario in (a) a physically embodied agent; (b) a network-deployed MAS; (c) an agent-based simulation.

a future perception on its own position will return the updated position. In *IRL*, a similar call results in an actual movement of the agent's body in the physical environment.

The different relations between the agent decision process – the agent mind – and other components in a scenario are shown in Figure 1, for three cases: a physical agent body that the agent mind controls; a MAS in which the agent mind interacts directly with the framework or with the local system, via the programming language, as in Jade; and an agent-based simulation, where the agent mind interacts with the simulation framework and its execution is controlled externally.

Regarding communication, agents must be able to communicate directly, via one-to-one messages, or via broadcasting in a given vicinity. Such communicative acts must be perceived as such by other agents. Agents can also communicate indirectly, but this can be covered by the relation of the agents to the environment. In *Sim*, communication can many times be performed based on vicinity (e.g. send a message to all agents of a certain type and in a given range). In *IRL*, communication is either direct and identifier-based, or based on artifacts and/or

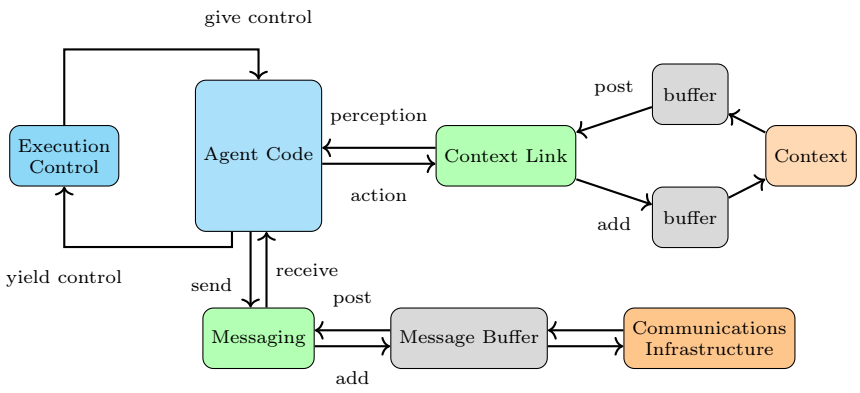


Fig. 2. A general perspective on how the agent code interacts with other components that ensure the separation from the environment and the communications infrastructure. The diagram applies especially to *Sim* scenarios, where interactions need to be buffered to ensure the appearance of simultaneity.

organizations which allow broadcasting. *IRL* applications need an actual means of discovery and communication, such as unique identifiers, a registration server and a communication protocol (XMPP in SPADE, or TCP/IP in Jade).

To ensure the same agent code can run in both *Sim* and *IRL*, we need to represent the relation with the environment in the same manner in both scenarios. Based on the principles in the A&A approach, the modeling used in the Repast framework, and our own previous work in MAS modeling [21], we abstract the elements of the environment as *contexts*. Any agent can be part of any number of contexts, and a context can offer functionalities, e.g. positioning and movement of agents.

For instance, a *spatial context* may allow placement of the agent in a topology, movement of agents in the topology, and observation of the surroundings; an *agent management* context may allow the creation or destruction of agents; a *temporal context* may allow the agent to know what time it is (in simulation / application time) and to schedule actions and behaviors in the future; and so on.

In MAS, direct communication is seen as a special functionality, separate from the interaction with the environment, so, while we can perceive it as a *communication context*, we will give it a special place.

One important difference between *Sim* and *IRL* applications is the execution. In simulations, agents run in a step-wise manner or use synchronization events, and the speed of the simulation can be controlled by the user. In network deployments, agents usually run in their own threads and are synchronized only loosely, via communication, many times using timers or other mechanisms linked to the time of the system, rather than the time of the simulation. Indeed, frameworks such as Jade or SPADE incentivize the use of behaviors as a means of internal organization of agents, such that the framework has some control over

the execution and *could* call behaviors at a controlled pace, but behaviors can still contain blocking calls, timers, or sleep calls.

3.2 Principles

We have derived the following principles that are necessary to ensure that the same agent code can run in both *Sim* and *IRL*. The components that we highlight in the principles are shown in Figure 2. The three principles are:

P1 The simulated environment is represented by a set of *simulation contexts*. Real-life environments may or may not be implemented as contexts, but still they should be *accessed* by agents as such. All *actions* executed by the agents (*external* actions) must be performed via such a dedicated interface. Similarly, all *perceptions* are obtained via a context. These interfaces must not necessarily be one single interface, but they must be *external to* the agent's decision process. We call the set of interfaces the *Context Link*. The *Context Link* must also handle any perception of time, system access, random numbers, etc.

P2 Similarly to external actions, communications and interactions between agents must go through a component external to the agent's decision process. Moreover, in *Sim*, communications must follow the time of the simulation and must replicate any latencies or reliability issues that are present in the *IRL* scenario.

P3 The execution model of agents must fully support either step-wise execution or event-based simulation control. In *Sim*, given that there will exist an external executor, the agent be able to *yield control* a short time after it *receives control*. As such, there are two possibilities for any agent: step-based execution or event-based execution control.

In **step-based execution**, the execution of the agent happens in small, self-contained *steps*: everything that happens inside the step must run independently from other agents, and any interaction – with the environment or with other agents – must take place between one step and the next. From the perspective of the agent, the *decision* takes place inside a step, during which the agent posts any interactions it wishes to initiate, and the *interactions* take place between steps (see Figure 6). Since no actions or communications can be expected to take place instantaneously, and also in *IRL* scenarios the agent needs to wait for the interaction tu actually happen, from the point of the agent it cannot *perceive* how much time passed between steps, as long as interactions between all agents happen at the same pace.

In **event-based control**, the execution of an agent can be interrupted and resumed at any time, with fine granularity, via events such as *suspend* and *resume*. For this to be achieved, the agent must handle events with promptness. The execution time allowed between a *suspend* and a *resume* event (equivalent to a *step*), should be of the same order of magnitude as the time to execute an action, or to send a message, such that in a step the agent will expect to receive the responses to messages sent in the previous step.

```
to go
rt random 50
lt random 50
fd 1
...
let prey one-of sheep-here
if prey != nobody [
  ask prey [ die ]
]
end
```

(a)

```
public void step() {
Context context = ContextUtils.getContext(this);
Grid patch = (Grid) context.getProjection("Simple Grid");
  ⟨compute new x, y⟩
patch.moveTo(this, x, y);
for(Object o: patch.getObjectsAt(x,y))
  if(o instanceof Sheep)
    (Sheep)o.die();
}
```

(b)

```
public void setup() {
addBehavior(new TickerBehavior(this, tick) {
x = ⟨new x⟩; y = ⟨new y⟩;
DFAgentDescription template = new
  DFAgentDescription();
ServiceDescription sd = new ServiceDescription();
sd.setType("sheep");
sd.addProperties(new Property("x", x));
sd.addProperties(new Property("y", y));
template.addServices(sd);
DFAgentDescription[] result =
  DFService.search(this, template);
if (result.length > 0) {
  ACLMessage msg = new ACLMessage(REQUEST);
  msg.addReceiver(result[0].getName());
  msg.setContent("hunt");
  send(msg);
} }); }
```

(c)

```
public void step() {
Position pos = e.getCurrentPosition();
for(Entity en : e.getEntitiesAt(pos))
  if(en instanceof SheepAgent)
    e.requestDestroyAgent(en);
List<Position> next =
  e.getFreeNeighborPositions(pos)
Position newPos =
 next.get(random.nextInt(next.size()));
e.moveToPosition(newPos);
}
```

(d)

Fig. 3. The code for the *Wolf* agent in the predator-prey scenario in NetLogo (a), Repast (b), Jade (c), and Flash-mas with ABM adaptations (d).

If these principles are followed, they are *sufficient* to ensure compatibility between *Sim* and *IRL* scenarios. Agent code from a *IRL* application, following the principles, can be *simulated*, because all interaction with its exterior goes through contexts or through communications support, and its execution can be controlled. If all agents decide on their actions and interactions in the *step phase*, and all agent actions and interactions are carried out in the *update phase*, which is disjoint from the *step phase*, then it will *appear* to agents that actions happen simultaneously, as shown in Figures 6 and 4.

Conversely, agent code from a *Sim* application, following the principles, can be *deployed* in real life, as all of its external actions can be directed to its body, its communications can go through an appropriate support infrastructure, and it can be executed by an execution engine either calling its `step` method, or running it and not ever suspending it.

The principles are also *necessary* to ensure compatibility, forming a minimal set. None of the principles could be ablated.

Should P1 not be respected, and should the agent be able to execute external actions directly, in simulation it may perform multiple actions in the same step, while the other agents are waiting their turn, breaking the illusion of simultaneity of agents' execution.

```
1. start all entities
2. for each step
3. | push events from each simulation      → no interactions between simulation objects
context to each simulation object          added here, but internal state can be updated
4. | step() each simulation object         → objects register pending actions to the contexts
5. | simulation contexts execute actions   → this is when actions are actually executed
```

Fig. 4. High-level view of the process of executing simulation objects.

Should P2 not be respected, and should agents be able to communicate or interact directly, in simulation again they would break the illusion of simultaneity, and it would not be possible to use an appropriate communications infrastructure for each situation.

Finally, should P3 not be respected, controlling the execution of the agent would not be possible. Inability to suspend execution of an agent would mean the inability to practically simulate large numbers of agents on a single system. If agents do not respect the length of a step, then some agents would be able to do more things in one global step than others, again breaking perceived simultaneity.

3.3 Execution

The discussion about execution is important because while in MAS frameworks and network-distributed deployments agents control their own execution or the execution is organized in behaviors, whereas in ABM frameworks execution is usually step-wise (tick-based) or reflex-based (in reaction to events)[1].

However, completely isolating the agent mind from the point of view of interaction with the environment, communications, and execution, means that the pace of execution could be controlled. To show this, we consider three types of internal agent architectures: *step-wise* – like in most ABM applications, in which the decision making policy is contained in a `step()` method which performs one unit of the agent's activity, with all agents performing roughly the same amount of processing in one step; *behavior-based* – like in JADE or SPADE, in which the agent is defined by a set of behaviors that react to events (e.g. messages) or are scheduled, with the behaviors themselves not blocking the agent and not taking more than a reasonable amount of time; and *event-driven* – like, for instance, in FLASH-MAS, in which the agent is built around a queue of events, which it processes in order, one at a time.

In any case, according to principles P1 and P2, performing any action (including communicative acts) is done via a context, which, depending on the type of scenario (*IRL* or *Sim*) performs the action *immediately* or *defers* it to a later time.

If the three principles are respected, execution can be *suspended*, and execution speed can be *controlled*. The execution is detailed in Figure 5. Since the realization of actions is decoupled from the time at which the action is requested in the agent code, then the following conditions are obtained:

[1] GAMA also supports equation-driven dynamics.

```
Simulation_step()
global: step_time
initially: step ← 0 ; time[a] ← 0, ∀a ∈ agents ;
    pending_messages ← ∅ ; deliverable_messages ← ∅
step += 1
for a ∈ agents do                                                  ____________
  for c ∈ contexts                                                  update phase
    c.sendEventsTo(a)        ⟨includes messages from deliverable_messages⟩
      ⟨agents process events, update state, but initiate no actions⟩
for a ∈ filter_active(agents) do                                   ____________
  switch a.type :                                                   agent step phase
    case step-wise:
      a.step()
        ⟨pulls events and messages not received at step 4⟩
        ⟨actions submitted to contexts, as pending⟩
        ⟨messages added as pending in pending_messages⟩
    case behavior-based:
      while time[a] < step * step_time
        b ← a.next_behavior()
          ⟨actions and messages added as pending⟩
        Δt ← measure-time: b.run()
        time[a] += Δt
    case event-driven:
      if time[a] < step * step_time
        @time ti : a.receive_event( RESUME )
          ⟨processes any events arrived since last SUSPEND⟩
          ⟨actions and messages added as pending⟩
        sleep(step * step_time − time[a])
        a.receive_event( SUSPEND )
        wait until not a.is_running()
        @time tf : time[a] ← time[a] + tf − ti                     ____________
for c ∈ contexts                                                    apply actions phase
  c.applyActions()    ⟨clears pending actions⟩
m ← filter_deliverable_messages(pending_messages)
deliverable_messages ← deliverable_messages ∪ m
pending_messages ← pending_messages \ m
```

Fig. 5. Pseudo-code for one step / one time unit of the simulation, with cases for each type of internal agent architecture. We use **measure-time** and **@time** constructs to signify measurement of the time for a statement to complete and the time at which a statement gets to run, respectively.

- no actions are *actually* executed during the *agent step phase* (the state of the environment is not changed), so during their activity all agents *perceive* the same environment, before any action from that step is applied;
- all actions are realized (applied to the environment) during the *update phase*, providing the appearance of simultaneity;
- subject to cooperation from the agent code (behaviors / event processing do not block or take disproportionately longer than other agents), each agent step takes more or less the same amount of time, *chosen* to be enough for the decision process but also sufficiently granular for sequential interactions to appear natural – during a step an agent can perform about one action, or can send about one message in a conversation;
- no messages are delivered instantaneously, and message delivery for any message takes at least one step, and may take more depending on how the function $filter\_deliverable\_messages$ is implemented;

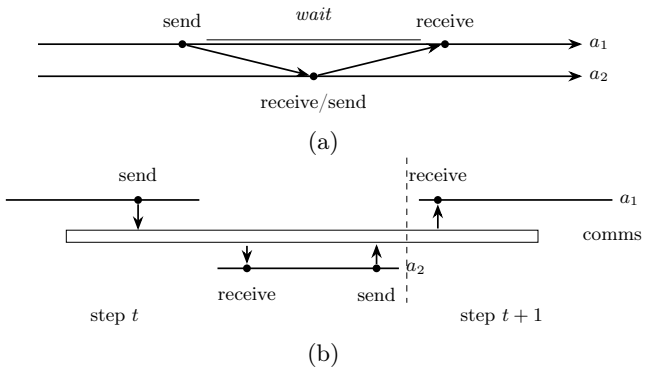


Fig. 6. (a) a round-trip message between agents a_1 and a_2 in *IRL* or in a *Sim* scenario, as *perceived* by the agents. (b) the same interaction, in step-based execution.

– for *event-driven* agents, messages and other events *can* be delivered at any time, as the agent is suspended and will only be able to process them after the RESUME event.

When agents run in *IRL* scenarios, they can be executed independently, in their own threads, and interaction can take place in the same manner as in *Sim* scenarios, just that they do not need to be buffered any more. The *Context Link* components, adapted to the *IRL* scenario, can relay the perceptions and the actions directly from and to the appropriate external components, without buffering. Similarly, messages can be relayed to the agent as they are received via the *Communications Infrastructure*.

4 Proof-of-Concept Modeling and Implementation

We have performed a primary validation of the principles presented in the previous section by creating a proof-of-concept implementation of a classic ABM scenario – the predator-prey scenario. In this scenario, a number of *wolf* agents and a number of *sheep* agents share the same discrete environment in which sheep consume grass and wolves eat sheep.

4.1 Prerequisites

We have validated the principles by implementing the scenario as an ABM simulation in a framework that was initially built for *IRL* deployments. We have chosen for this implementation the FLASH-MAS framework, not only because it is built in our team and we have more control over it, but because its flexibility and modularity make the implementation much easier[2].

[2] FLASH-MAS is open source and available at https://github.com/andreiolaru-ro/FLASH-MAS. This work is based on the `abms` branch, due to pe merged during 2026.

Flash-mas is a MAS framework built around the idea that all persistent elements in a MAS are *entities* sharing a uniform interface [19]. Entities, some of which can be distributed, are placed in the context of one another and are able to access the functionality of their containers. In Flash-mas, each agent normally (*IRL*) has its own execution thread, in which a queue of events is processed and disseminated to sub-agent entities called *shards*, each shard managing an aspect of the agent's functionality, such as messaging, remote operation, graphical interface, goal management, etc. Among other predefined entities there are the *nodes* – representing the presence of the framework on a machine – and *pylons* – representing communication infrastructures that span across multiple nodes. However, any custom type of entity can be defined and integrated seamlessly into the framework.

Central to Flash-mas is that entities are placed *in the context of* one another, such that each entity has access to the functionality of entities in the context of which it is placed. For instance, agents, but also their shards, have access to the pylons that contain them. Or, if an agent is placed in the context of a node, its shards have access to functionality offered by the node, etc.

4.2 Components

For the implementation, we have asked the question: *can we run a simulation in a framework primarily built for network-deployed applications and, if yes, what are the components needed to be added in the framework to support ABM simulation*?

Following the entity-based principle of the Flash-mas framework, we have first identified which are the different types of entities needed to run an ABM simulation. The criteria by which we have decided to separate components (rather than just have a monolithic approach) was which components could we want to be able to implement differently, or customize for different functionality, while leaving the others untouched. As such, we have identified several **types of entities**: the *simulation*, the *executor*, *simulation contexts*, *environment link shards*, and *reporter*. The relations between these types of entities are summarized in Table 1.

The **simulation** takes the place of the *node* in an *IRL* deployment to manage the simulation and the entities within it. Beside the executor, the simulation contains the *simulation objects* – the entities that are *inside* the simulation, such as agents – and *simulation contexts* – the entities which are not inside the simulation, and that represent the environment of the simulated objects.

The **executor** handles execution control, such as stepping the entities that support step-wise execution, and suspending and resuming entities that do not.

Simulation contexts are any entities that represent the environment and anything that is *outside* of the agent's mind. There are several simulation contexts that are general to many ABM applications, such as the spatial context (*where* the agents are located), the temporal context (*when* things should happen in the simulation), social context (*who* interacts with an agent), and agent management context (creating and destroying agents).

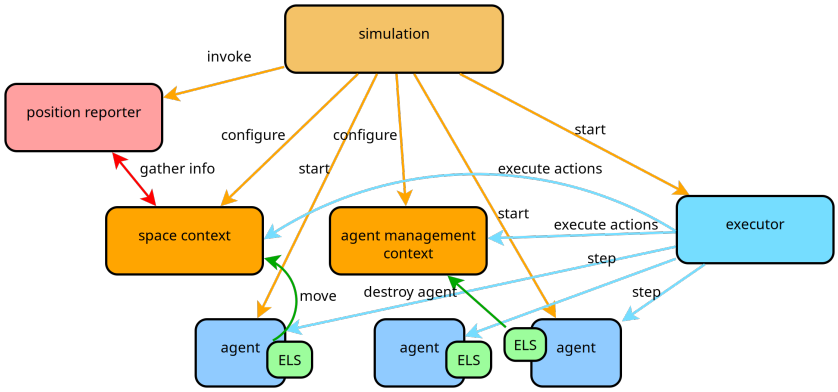


Fig. 7. The entities and relations among them in a simulation setup. *Agents* use *environment link shards* to connect to *contexts*. The *simulation* creates and configures all other entities. The *executor* steps the agents and instructs the execution of actions to the contexts. The *reporter* obtains information from the contexts to be visualized.

When an agent or another entity requests a context to perform an action, the action is added to a *queue* and performed when the executor signals that all the agents have completed their step (see Figure 5). Since the agents are not able to obtain an *immediate* feedback for their actions, contexts *store* meta-data for the actions, such as the status (pending / completed / failed), action result, and the originating entity. The status and the result of the action can be checked by the agent later on.

Simulation objects are any entities that are an *object* of the simulation. They are part of the simulation and are being studied through it. These are usually agents, but there could be other entities as well. For instance, to eschew the need to represent patches as agents, as in NetLogo, patches may be their own type of first-class entities.

An **Environment Link Shard** (*ELS*) is an entity that is part of an agent or another simulated object and ensures the link between the entity and the various contexts that is a part of. While the implementation supports multiple ELS for the same simulation object, it is easier to just use one for the usual cases. The reason why we should have the ELS as an interface between the simulated object and the simulation contexts is twofold: (1) to separate from the agent's mind the concern of discovering and managing the link with the simulation contexts and to offer a single point of relation to the contexts; and (2) to ensure that the agent code can use the same constructs, that access the functionality of the ELS, without concern for how exactly the functionality of the contexts is accessed, offering flexibility when moving between the *Sim* and *IRL* scenarios. We will show some examples below.

Reporters are entities that allow the visualization and reporting on the state of simulated objects. They are similar to contexts, but they do not offer functionality to the simulated objects and are not visible to them, rather they

Entity	Direct context	Other context	How context is used
simulation	–	–	–
executor	simulation	–	access to the list of entities in the simulation
simulation context (*sim-context*)	executor	simulation	simulation: access to other simulation contexts, access to list of simulation objects, access to execution status
simulation object (*sim-object*)	sim-context	executor, simulation	sim-context: access to environment and communication
agent group (is a *sim-object*)	sim-context	executor, simulation	sim-context: access to environment and communication
agent (is a *sim-object*)	agent group *or* sim-context	executor, simulation	sim-context: access top environment and communication; agent group: coordination with other agents in the group
shard	agent	sim-context, executor, simulation	agent: access to other shards; sim-context: access to environment and communication
ELS	sim-object	sim-context, executor, simulation	sim-context: access to environment and communication
reporter	executor	simulation	simulation: access to data avout sim-contexts and sim-objects

Table 1. Relations between types of entities and their contexts. A is a direct context of B if no other entity C is a context of B and in the context of A.

obtain information about the objects via the contexts or directly via accessible properties of the objects.

Agent Groups are a helper entity that enables the loading and management of groups of identical agents. The group *loads* the agents and configures them, following rules specific to the group.

The implementation uses the existing Flash-mas **communications pylon** abstraction as a provider for communication services, and the *messaging shard* as interface to the pylon. In *IRL* scenarios, messages will be delivered immediately; in *Sim* scenarios, messages are kept as *pending* and delivered in the next step.

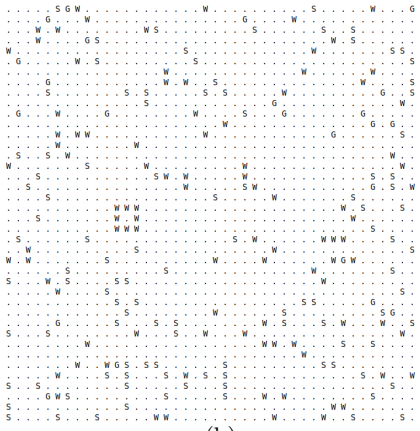
4.3 Deployment and Functionality

To validate whether a MAS framework can run ABM simulations, and how agent code can be switched between *Sim* and *IRL* scenarios, we have implemented several applications using Flash-mas and the *simulation contexts* that we have developed. Among these, a classic wolf-sheep scenario and an application for the dynamic allocation of meeting rooms.

Deployment of these scenarios uses the hierarchical, entity-type-based configuration in Flash-mas. Configuration is similar in the two types of scenarios, except for simulation contexts and communication platforms in *Sim* and *IRL* scenarios, respectively.

```
-load_order simulation;executor;context
-package abms.wolfSheepPredation
-simulation sim classpath:Simulation
  -executor StepWise:StepWise steps:100
    -context AgentManagement:agentManagement
    -context Space:space width:100 height:100
    -context Random:random seed:64
    -context ProximityCommunication:comms
      -WolfSheepGroup g sheepCount:5000
                        wolfCount:2000
                        grassCount:500
```

(a)



(b)

```
-load_order simulation;executor;context
-package abms.smartMeeting
-simulation sim classpath:Simulation
  -executor StepWise:StepWise steps:60
    -context AgentManagement:agentManagement
    -context Random:random seed:51
    -context GraphCommunication:communication
    -context Space:space topology:graph
      nodes:lobby,r1,r2,r3,r4,r5,r6
      edges:lobby-r1,lobby-r2,lobby-r3,r1-r4,r3-r5,r5-r6
    -SmartMeetingGroup g
      -agent Auction n:1 releaseAfterSteps:20
      -agent Room n:6 capacity:8
        equipment:PROJECTOR,WHITEBOARD,VIDEO_CONFERENCE
      -agent Person n:3 auctionAgent:auction0
```

(c)

```
-package abms.smartMeeting
-node lobby
  -pylon WSRegions:p-lobby
    -agent Auction:action0
        releaseAfterSteps:20
    -agent Room:lobby
    -agent Person:p1
        auctionAgent:auction0
-node r1
  -pylon WSRegions:p-r1
    -agent Room:r1 capacity:8
        equipment:PROJECTOR
-node r2
  -pylon WSRegions:p-r2
    -agent Room:r2 capacity:8
        equipment:WHITEBOARD
...
```

(d)

Fig. 8. Deployment configurations in Flash-mas for various scenarios: (a) the wolf-sheep multi-agent system in a *Sim* scenario; (b) simple visualization for the wolf-sheep scenario, cropped from a 7000 agent deployment; (c), (d) deployment configurations for the smart room scenario, in *Sim* and *IRL*, respectively.

The simulation contexts that we have developed to support *Sim* scenarios can be *reused* between applications, for instance the `AgentManagement` and `Random` contexts, as well as the `Space` context, even if using different *topologies*.

The wolf-sheep simulation has been deployed with 5000 sheep and 2000 wolves, to evaluate if increasing the number of agents has a significant overhead. As execution of the agents was sequential, no problems were encountered. Further testing will be performed with larger numbers of agents and more complex behaviors, studying the impact of the number of agents on the performance of the simulation.

5 Discussion

Even if the principles presented in Section 3 have been developed while designing and implementing simulation features in the Flash-mas framework, they have been created as general guidelines that could be applied in the case of other MAS frameworks.

In JaCaMo-sim, the decoupling of agent code from the environment is done via Execution Contexts. Moreover, JaCaMo has the capability to run on different underlying communication infrastructures, both Jade-based (slow, but network-capable) and local (fast, but constrained to one node). However, as artifacts are programmed in Java, and not as higher-level entities, their execution cannot be controlled as well as for agents.

Jade would require several changes to become capable of local simulations. More flexibility is required from changing the message transport protocol. Behavior execution can be changed to allow the control of behavior execution. Agents are already isolated from their executors (the developer does not create the thread that the agent runs on), so from this point of view simulation would be possible. However, as highlighted by [10], blocking calls are an issue, as well as some of the temporal management. Jade has not been built as a modular framework, which makes the replacement of the framework components difficult.

Conversely, enabling the running of ABM-specific agent code in real-life deployments also brings challenges. In NetLogo and GAMA, the use of specific programming languages hinders interoperation with outside components, as they would need the addition of dedicated language constructs.

Repast, with agents written in Java and with its reliance on contexts, comes closest to having the ability to deploy agents in a real-life scenario, with the condition, of course, of building adequate interaction infrastructures and environment abstractions.

6 Conclusions and Future Work

We have observed that reconciling ABM platforms with MAS frameworks cannot be achieved, when agents are programmed in a general-purpose programming language, without code respecting several principles when interacting with the environment, timing, and communicating with other agents, which must be done via dedicated components. Agent code must allow for execution control.

This research has a long way to go. We envision bringing simulation capabilities to applications using agents programmed in domain-specific languages, such as ASTRA or Jason. We plan a series of complex scenario deployments to test the limits of our approach, as well as the development of visualization and monitoring tools that allow for the quantitative evaluation of these scenarios.

Acknowledgement. This research is supported by the project "Romanian Hub for Artificial Intelligence – HRIA", Smart Growth, Digitization and Financial Instruments Program, 2021-2027, MySMIS no. 334906.

References

1. Al-Zinati, M., Zalila-Wenkstern, R.: Matisse 2.0: A large-scale multi-agent simulation system for agent-based its. In: 2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT). vol. 2, pp. 328–335. IEEE (2015)
2. Al-Zinati, M., Zalila-Wenkstern, R.: Divas: a multi-agent simulation framework. International Journal of Agent-Oriented Software Engineering **7**(2), 152–183 (2022)
3. Baiardi, M., Burattini, S., Ciatto, G., Pianini, D.: Testing bdi-based multi-agent systems using discrete event simulation. Autonomous Agents and Multi-Agent Systems **40**(1), 18 (2026)
4. Barone, D.A., Wickboldt, J.A., Cavalcanti, M.C.R., Moura, D.F., Tesolin, J.C.C., Demori, A.M., dos Anjos, J.C., de Carvalho, L.F.B.S., Gomes, J.E.C., de Freitas, E.P.: Integrating a multi-agent system simulator and a network emulator to realistically exercise military network scenarios. In: SIMULTECH. pp. 194–201 (2023)
5. Batista Júnior, A.d.A., Coutinho, L.R.: Incorporating explicit coordination mechanisms by agents to obtain green waves. In: Advanced Methods and Technologies for Agent and Multi-Agent Systems, pp. 137–145. IOS Press (2013)
6. Bellifemine, F., Poggi, A., Rimassa, G.: JADE - a FIPA-compliant agent framework. In: Proceedings of PAAM. vol. 99, pp. 97–108. Citeseer (1999)
7. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Science of Computer Programming **78**(6), 747–761 (2013)
8. Briola, D., Vizzari, G., Montinaro, M., et al.: Enhancing testing of mass with a simulator of jade. FRONTIERS IN ARTIFICIAL INTELLIGENCE AND APPLICATIONS **413**, 3687–3694 (2025)
9. Candela, E., Parada, L., Marques, L., Georgescu, T.A., Demiris, Y., Angeloudis, P.: Transferring multi-agent reinforcement learning policies for autonomous driving using sim-to-real. In: 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 8814–8820. IEEE (2022)
10. Cardoso, H.L.: Sajas: enabling jade-based simulations. In: Transactions on Computational Collective Intelligence XX, pp. 158–178. Springer (2016)
11. Collier, N.: Repast: An extensible framework for agent simulation. The University of Chicago Social Science Research **36**, 2003 (2003)
12. Collier, R., Russell, S., Ghanadbashi, S., Golpayegani, F.: Towards the use of hypermedia mas and microservices for web scale agent-based simulation. SN Computer Science **3**(6), 510 (2022)
13. Galland, S., Rodriguez, S., Gaud, N.: Run-time environment for the SARL agent-programming language: the example of the janus platform. Future Generation Computer Systems **107**, 1105–1115 (2020)
14. Grosjean, L., Drogoul, A., Herrmann, B., Huynh, N.Q., Lang, C., Marilleau, N., Philippe, L.: Distribution model: Separation of concerns to facilitate the distribution of agent-based models. In: International Conference on Practical Applications of Agents and Multi-Agent Systems. pp. 105–117. Springer (2025)
15. Jagutis, M., Russell, S., Collier, R.: Flexible simulation of traffic with microservices, agents & rest. International Journal of Parallel, Emergent and Distributed Systems **38**(6), 490–506 (2023)
16. Khatiri, S., Panichella, S., Tonella, P.: Simulation-based testing of unmanned aerial vehicles with aerialist. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings. pp. 134–138 (2024)

17. Labiosa, A., Hanna, J.P.: Multi-robot collaboration through reinforcement learning and abstract simulation. In: 2025 IEEE International Conference on Robotics and Automation (ICRA). pp. 1838–1845. IEEE (2025)
18. Novischi, D.M., Florea, A.M.: Toward a real-time heterogeneous mobile robotic swarm: Robot platform and agent architecture. In: 2013 17th International Conference on System Theory, Control and Computing (ICSTCC). pp. 772–776. IEEE (2013)
19. Olaru, A., Sorici, A., Florea, A.M.: A flexible and lightweight agent deployment architecture. In: 2019 22nd International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania, 28-30 May 2019. pp. 251–258. IEEE (2019). https://doi.org/10.1109/CSCS.2019.00048, https://ieeexplore.ieee.org/abstract/document/8744845/
20. Olaru, A.: FLASH-MAS: towards a state-of-the-art,modular agent deployment environment. In: Proceedings of EUMAS 2025, The 22nd European Conference on Multi-Agent Systems, September 3-5, Bucharest, Romania (2025)
21. Olaru, A., Nicolae, G., Florea, A.M.: The entity-operation model for practical multi-entity deployment. In: Ciortea, A., Dastani, M., Luo, J. (eds.) Engineering Multi-Agent Systems. pp. 253–270. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-48539-8_17, https://link.springer.com/chapter/10.1007/978-3-031-48539-8_17
22. Palanca, J., Rincon, J.A., Carrascosa, C., Julian, V.J., Terrasa, A.: Flexible agent architecture: mixing reactive and deliberative behaviors in spade. Electronics **12**(3), 659 (2023)
23. Palanca, J., Terrasa, A., Julian, V., Carrascosa, C.: Spade 3: Supporting the new generation of multi-agent systems. IEEE Access **8**, 182537–182549 (2020)
24. Pinyoanuntapong, P., Pothuneedi, T., Balakrishnan, R., Lee, M., Chen, C., Wang, P.: Sim-to-real transfer in multi-agent reinforcement networking for federated edge computing. In: 2021 IEEE/ACM Symposium on Edge Computing (SEC). pp. 355–360. IEEE (2021)
25. Polhill, J.G., Ge, J., Hare, M.P., Matthews, K.B., Gimona, A., Salt, D., Yeluripati, J.: Crossing the chasm: a "tube-map" for agent-based social simulation of policy scenarios in spatially-distributed systems. GeoInformatica **23**(2), 169–199 (2019)
26. Rao, A.S., Georgeff, M.P.: BDI agents: From theory to practice. In: Lesser, V.R., Gasser, L. (eds.) Proceedings of the ICMAS-95, First International Conference on Multiagent Systems, June 12-14, San Francisco, California, USA. pp. 312–319. San Francisco, CA, The MIT Press (1995)
27. Ricci, A., Croatti, A., Bordini, R., Hübner, J., Boissier, O.: Exploiting simulation for mas programming and engineering-the jacamo-sim platform. In: 8th International Workshop on Engineering Multi-Agent Systems (EMAS 2020) (2020)
28. Singh, D., Padgham, L., Logan, B.: Integrating bdi agents with agent-based simulation platforms. Autonomous Agents and Multi-Agent Systems **30**(6), 1050–1071 (2016)
29. Taillandier, P., Vo, D.A., Amouroux, E., Drogoul, A.: GAMA: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In: International Conference on Principles and Practice of Multi-Agent Systems. pp. 242–258. Springer (2010)
30. Tiwari, R., Khapre, S., Singh, A.: Reinforcement learning in robotic systems: A review on sim-to-real transfer. Robotics and Autonomous Systems p. 105327 (2026)
31. Van Haare Heijmeijer, A., Alves, G.V.: Development of a middleware between sumo simulation tool and jacamo framework. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal **7**(2), 5–15 (2018)

32. Wilensky, U., Rand, W.: An introduction to agent-based modeling: modeling natural, social, and engineered complex systems with NetLogo. MIT press (2015)