# Addressing the Challenges of Conservative Event Synchronization for the SARL Agent-Programming Language

Glenn Cich[1⋆], Stéphane Galland[2], Luk Knapen[1], Ansar-Ul-Haque Yasar[1], Tom Bellemans[1], and Davy Janssens[1]

[1] Hasselt University, Transportation Research Institute (IMOB), Agoralaan, 3590 Diepenbeek, Belgium
[2] LE2I, Univ. Bourgogne Franche-Comté, UTBM, F-90010 Belfort, France

**Abstract.** Synchronization mechanism is a key component of an agent-based simulation model and platform. Conservative and optimistic models were proposed in the domain of distributed and parallel simulations. However, the SARL agent-programming language is not equipped with specific simulation features, including synchronization mechanisms. The goal of this paper is to propose a conservative synchronization model for the SARL language and its run-time platform Janus.

**Keywords:** Multi-agent simulation, Conservative event synchronization, SARL agent programming language, Janus platform

## 1 Introduction

In agent-based simulations, the entire simulation task is divided into a set of smaller sub tasks each executed by a different agent. These agents may run in parallel, and communicate with each other by exchanging timestamped events or messages. In this paper, an event refers to an update to the simulation system's state at a specific simulation time instant. Throughout the simulation, events arrive at destination agents, and depending on the delivery ordering system of the simulation, they are processed differently. The two commonly used orderings are (i) event reception order and (ii) event timestamp order (the timestamp is assigned to the event by the emitting agent). With the first type, events are delivered to the destination processes when they arrive at the destination. On the other hand, with the timestamp-order, events are delivered in non-decreasing order of their timestamp, requiring runtime checks and buffering to ensure such ordering.

The key question is how to create a synchronization model, and its implementation, based on the SARL agent-programming language, and assuming that the execution platform is fully distributed. In this paper, the SARL agent-programming language is equipped with an event synchronization model with

⋆ *Correspondence to*: glenn.cich@uhasselt.be

the following characteristics: (i) The model follows a conservative synchronization approach. (ii) The synchronization process is hidden to the agents by using the *capacity* and *skill* concepts. (iii) The agent environment is integrated into the synchronization process.

The remainder of the paper is organized as follows: in Sect. 2, an overview of the current state of the art is given. Section 3 introduces the agent based framework SARL that is used in the model described in this paper. Section 4 describes the method used to synchronize events in SARL. The evaluation of this method is described in Sect. 5. Finally, in Sect. 6, the paper is concluded.

## 2 Related Work

Parallel Discrete-Event Simulation (PDES) has received increasing interest as simulations become more time consuming and geographically distributed. A PDES consists of Logical Processes (LPs) acting as the simulation entities, which do not share any state variables (similar to agents) [3].

A PDES that exclusively supports interaction by exchanging timestamped messages obeys the local causality constraint if and only if each LP processes events in non-decreasing timestamp order [2]. To satisfy the local causality constraint, different synchronization techniques have been proposed for distributed systems which generally fall into two major classes of synchronization: *conservative or pessimistic*, which strictly avoids causality violations; and *optimistic*, which allows violations and recovers from them.

Conservative synchronization algorithms strictly avoid any occurrence of causality errors. To do so, the LP is blocked from further processing of events until it can make sure that the next event in its local future event list has a timestamp smaller than the arrival time of any event that might be arriving at the LP in the future. The main issue of any conservative parallel simulator is determining if it is safe for a processor to execute the next event. To deal with this issue, several techniques have been proposed which are further classified into four categories: methods with dead-lock avoidance, deadlock detection and recovery, synchronous operation, and conservative time windows.

Optimistic synchronization algorithms do not try to stop the LP's execution to synchronize them. It allows causality errors to occur and to be detected by the arrival of an event with a timestamp that is less than the local time of the receiving LP. Optimistic algorithms recover from the causality error by undoing the effects caused by those events processed speculatively during the previous computation. This recovery operation is known as *rollback*, during which the state of the LP is restored to the one that was saved just prior to the timestamp of the violating event. The main issue of any optimistic parallel simulator is related to the necessary storage space that is needed for recovery, and the positive ratio of the time spent for performing the recovery on the time spent for executing the behavior of the system.

In the past three decades, numerous approaches have been proposed by different researchers in this field. A number of surveys can be found in the literature which summarize both conservative and optimistic techniques [2, 3, 5, 9, 10, 12].

In multiagent systems, several models of synchronization were proposed. In this domain, agents are assimilated to LPs. Weyns and Holvoet [13] describe a conservative synchronization module in the multiagent system that is based on the composition of the $synchronization_a$ modules for each agent $a$. The approach of the model is to let synchronization be the natural consequence of situatedness of agents and not be part of the agents decision mechanism. This is reflected in the fact that the composition of a set of synchronized agents only depends on the actual perception of the agents. Such synchronization is based on the exchange of a structured set of synchronization messages.

Braubach et al. [1] propose a centralized service that has the role to manage the time evolution, and to notify the agent when the time is evolving. In this model, each agent notifies the time management service when it has finished its task for a given time period. This model is one of the most simple pessimistic synchronization algorithms. Its major drawback is related to the introduction of the centralized service that makes it harder and less efficient to distribute the agents over a computer network.

Xu et al. [15] propose an asynchronous conservative synchronization strategy for parallel agent-based traffic simulations. The authors propose to replace the global synchronization barrier in the multiagent system by a local synchronization strategy that enables agents to communicate individually and providing each of the agents with a heuristic for increasing the time-window look ahead in order to predict the next safe events.

## 3 SARL: an Agent-Oriented Programming Language

SARL[3] is a general-purpose agent-oriented programming language [11]. Such language should thus provide a reduced set of key concepts that focuses solely on the principles considered as essential to implement a multi-agent system. In this paper, four elements of the metamodel of SARL are used: Agent, Space, Capacity and Skill. These four concepts are explained below.

- **Agent:** An *Agent* is an autonomous entity having a set of skills to realize the capacities it exhibits. An agent has a set of built-in capacities considered essential to provide the commonly accepted competences of agents, such as autonomy, reactivity, proactivity and social capacities. The various behaviors of an agent communicate using an event-driven approach.
- **Space:** A *Space* is the abstraction to define an interaction space between agents or between agents and their environment, which may be the real world or a simulated environment. The simulated environment subsystem could be modeled with a multiagent system by itself. In the SARL toolkit, a concrete default space, which propagates events, called `EventSpace` is proposed.

[3] `http://www.sarl.io`

- **Capacity:** *A Capacity is the specification of a collection of functions* that support the agent's capabilities, which are represented by the Capacity concept. This specification makes no assumptions about its implementation. It could be used to specify what an agent can do, i.e. what a behavior requires for its execution.
- **Skill:** A *Skill* is a possible implementation of a capacity fulfilling all the constraints of this specification.

The Janus platform[4] was redesigned and reimplemented in order to serve as the software execution environment of the SARL programs. Janus is designed in order to be a fully distributed platform over threads and a computer network. The execution unit in Janus is the event handler: the part of the SARL agent that is executed when a specific event is received. Each of these units are executed in parallel to the other units, even in the same agent.

The design of the Janus platform may cause issues for creating agent-based simulation applications. Indeed, several notions of time must be considered: user time (the real time, machine time) and simulated time. According to Lamport [7], simulated time is a logical clock that induces a partial ordering of events; it has been refined in distributed context as logical virtual time by Jefferson [6]. The Janus platform does not make any assumption on the ordering of the events that are exchanged by the agents. As a consequence it is impossible to use the Janus platform for agent-based simulation involving a time concept without providing the platform with a specific synchronization mechanism. A model of such a mechanism is described in Sect. 4. Agents timestamp the event notifications they emit using their current perception of simulated time (the logical clock). They perceive each other behavior as a sequence of events ordered by the logical clock. Agents can only emit events that comply to the Lamport partial order for logical time induced the causality rule. In case agent $A_0$ uses information about agent $A_1$ notified by an event $E_0$ timestamped by $t(E_0)$, it can no longer notify any event $E_1$ that precedes $E_0$ in the partial order. This requires agents to synchronize their perception of the common logical clock.

Additionally, agent-based systems often include an *agent environment* , which is the software layer between the external world and the agents. This environment contains objects and resources, a.k.a. artifacts, that are not agents, but could be used by them. All the actions on the artifacts must be also synchronized in order to preserve the integrity of the agent environment state.

## 4 Event Synchronization Model for SARL

In this section, an event synchronization model for the SARL programming language and its Janus execution environment is presented.

A *time period* is delimited by two discrete moments in (real or simulated) time. Each *moment in time* can be thought to bear a label which is the *timestamp*. In the remaining part of the paper the terms *timestamp* and time period

[4] `http://www.janusproject.io`

will be used interchangeably. Hence, the term *timestamp* is also used to identify the time period starting at the *moment in time* it is associated with.

According to the SARL metamodel, interaction among the simulation agents on one hand, and between the simulation agents and the agent environment on the other hand is supported by events. Each event $e$ in the set $\mathbb{E}$ of events that are not already fired in the simulation agents is defined by a time stamped $t_e$ and a content $c_e$. The timestamp $t_e$ is the simulation time for which the event is fired. It is always greater or equal to the current simulation time $t$: $\forall e \in \mathbb{E}, e = \langle t_e, c_e \rangle \implies t_e \geq t$.

### 4.1 General Architecture

The proposed event synchronization model is designed by considering the following three major assumptions and constraints (in bold face).

The **synchronization process is hidden to the agents** by using the capacity and skill concepts. Indeed, the synchronization process is related to the simulation and not to the simulation agent architectures and models. For example, the simulation agent models should be the same if they are instantiated during simulation or deployed on embedded computers. In order to enforce this characteristic, we propose to provide skills that are implementing the standard interaction agent capability, which is provided by the SARL metamodel, with the proposed synchronization mechanisms. This approach enables a clear distinction between the application-dependent models in the agents, and the simulator-dependent modules. It increases the level of abstraction that the framework will provide to application developers.

The **agent environment is part of the simulated system**. The agent environment as a key component of the system must be considered in the event synchronization model. In this work, we assume that the agent environment is modeled with a complex hierarchy of agents, as proposed by Galland and Gaud [4]. The root agent in this hierarchy represents the entire environment for the application logic layer (even if the environment is distributed over multiple environmental agents). In the context of this paper, and for simplicity reasons, we make use of two kinds of agents: (i) an *environment agent*, and (ii) a *simulation agent*, which represents the application logic's agent.

The **event synchronization model follows a conservative synchronization approach**. As explained in Sect. 2, two major approaches of synchronization can be considered: conservative and optimistic. In order to select the best approach, we have considered the two types of interaction between the simulation agents and the environment: (i) the simulation agents perceive the state of the environment; and (ii) the simulation agent acts in order to change the state of the environment. First, consider the data representing the perception of an agent at simulation time $t$: this needs to be extracted from the same state of the environment for all agents in order to ensure the consistency of the agents' behaviors for time $t$. Second, the simulation agents are supposed to act in the environment simultaneously and autonomously. Solving the joint actions of the agents requires to avoid them to directly change the environment's state. Agents

are sending desires of actions, named *influences*, that are gathered and used by the agent environment in order to compute its next state. This approach is known as the influence-reaction model [8]. Because the agent environment may be modeled by means of a hierarchy of agents, according to Galland and Gaud [4], the influence-reaction model may be locally applied if each subagent inside the environment is supporting a specific spatial zone. The influence-reaction model implies the introduction of at least one rendez-vous point during the simulation process: the agent environment is waiting for all the simulation agents to provide their influences. Besides the types of interaction, one can take into account the possible drawbacks of an optimistic approach. The optimistic approach will need a lot of space in order to store the different states of the simulation. This indicates as well that this approach will be application dependent because the used data structures are application specific which might induce burden to the designer/programmer. The type of applications we have in mind need a strict synchronization between a lot of agents. The possibility of rollbacks will be very high and hence very time consuming. The bottle neck we introduce with our conservative mechanism will probably cause less time loss in these cases. These considerations lead us to select a conservative approach in designing our event synchronization model.

**Synchronization-Unaware Simulation Agent Architecture.** The general architecture for the simulation agents can be described by Algorithm 1.1. The simulation agents are able to react to `PerceptionEvent` events, which are fired by the agent environment to notify the simulation agent that its perception has changed. When the simulation agent has executed its reaction behavior, it sends its list of desired actions to the agent environment by calling the `influence` function. This function is provided by the `EnvironmentInteractionCapacity` capacity (Algorithm 1.1), which represents the capacity of an agent to interact with its environment. The `EnvironmentInteractionCapacity` implementation will pack the influences into an occurrence of the `AgentIsReadyEvent` event, and send the latter to the agent environment. The simulation agent is also able to react to events that were fired by other simulation agents.

```
agent SimulationAgent {
  uses EnvironmentInteractionCapacity
  on PerceptionEvent {
    /* React on the perception receiving from the environment */
    [...]
    /* Send AgentIsReadyEvent to the environment */
    influence([...])
  }
  on Event {
    /* React on events from simulation agents */
    [...]
  }
}
capacity EnvironmentInteractionCapacity {
  def influence(desiredActions : Object*)
}
```

**Algorithm 1.1.** General algorithm for the simulation agents and definition of the `EnvironmentInteractionCapacity` capacity.

All the agents in the SARL specification are provided with built-in capacities for which the execution platform provides the implementation. The first built-in capacity that is relevant to our synchronization model is `Time`. It provides the functions for accessing the value of the current simulation time $t$. The second built-in capacity is `Behaviors`. It provides the `asEventListener` function, which replies the entry point for all events that are received by the agent. This capacity also provides the `wake` function to emit events inside the context of the agent itself. Specific implementation of these two capacities will be provided in Sect. 4.2 in order to integrate our synchronization model in a way that is transparent to the simulation agent.

**Synchronization-Unaware Environment Architecture.** The general architecture for the simulation agents can be described by Algorithm 1.2. In this paper, we consider that the agent environment can be modelled using a dedicated (holonic) agent according to the model proposed by Galland and Gaud [4], in which the proposed agent represents the agent environment and is managing time evolution.

```
agent AgentEnvironment {
  uses TimeManager
  var expectedNumberOfInfluences : Integer
  var influences : List
  on StartSimulationStep {
    sendPerceptionsToAgents
  }
  on AgentIsReadyEvent {
    influences += occurrence
    if (influences.size == expectedNumberOfInfluences) {
      appliesInfluencesToEnvironmentState
      readyForTimeEvolution
    }
  }
}
```

**Algorithm 1.2.** General algorithm for the agent environment.

The agent environment is waiting for the `StartSimulationStep` event that is fired by the platform's time manager[5]. When the event is received, the agent environment computes the agents' perception from the environment's state and sends `PerceptionEvent` events to the simulation agents. The implementation of the `sendPerceptionsToAgents` is application specific; it is not detailed in this paper. When receiving the `PerceptionEvent` occurrence, each simulation agent updates its knowledge with the timestamp of the event.

After sending the perception to the simulation agents, the agent environment is waiting for the agents' influences, according to the influence-reaction model [8]. When all the expected influences are received, the agent environment updates its state, and notifies the time manager that the simulation time $t$ can evolve. Indeed, inside a simulation process including an environment as a whole entity, the simulation agent at time $t$ can evolve according to the state of the

[5] The time manager is a platform module or another agent that is storing and managing the time $t$ over the simulation.

environment [8, 14]. Basically, time evolution might be modeled by $t' := t + \Delta t$, where $t$ is the current simulation time, $\Delta t$ is a constant time evolution amount, and $t'$ is the new simulation time.

Additionally, we have considered to dynamically determine the time increment using $t' := \min\{t_e | \forall e \in \mathbb{E}, t_e > t\}$. This approach is still vulnerable to deadlocks of simulation agents when they enter a deadlock or unexpectedly crash. In this case, the agent environment will wait infinitely for their response and hence the simulation will end in a deadlock as well. However this issue can be solved by using the machine time in order to detect a deadlock. The environment agent could keep track of the expected execution time per agent. When an agent exceeds this time, the environment agent could assume there is something wrong; the environment agent can proceed and ask the agent in deadlock to leave the simulation. In case a simulation agent wants to leave the simulation (deliberately stop, or crash) the environment agent is aware of that by listening the specific events fired by the execution platform and can update its list of agents to monitor.

### 4.2 Conservative Event Synchronization Mechanism

The event buffering is needed to ensure a pessimistic approach. The main idea is that simulation agents can send events to each other, but the events are not directly fired to the appropriate simulation agent. Hence, if a simulation agent decides to send an event to another simulation agent, this event is saved somewhere within the agent. This is done for every event that is sent for a given time period.

In the SARL specification, events may be received by an agent from another agent or from itself. In the first case, the `Behaviors` built-in capacity provides the agent's event listener that could be used for receiving the events. For supporting the second case, the `Behaviors` capacity provides the `wake` function for firing an event inside the context of the agent. For every simulation agent, the events that are received from other agents, or from itself are intercepted by a specific skill implementation of the `Behaviors` capacity. The intercepted events are kept in a bucket until a specific event of type `TimeStepEvent` that is representing a time step in the simulation is received. The `PerceptionEvent` event described in the previous section is a subtype of `TimeStepEvent`. Consequently, when a simulation agent receives its perception from the agent environment, all the buffered events for the current simulation time are fired in the agent context as well as the `PerceptionEvent` whose occurrence advances the agent's time to the next timestamp. Algorithm 1.3 provides a SARL implementation of the specific skill. The internal class `InternalBuffer` is defined to represent the event buffer (defined as a multiple-value map).

If the received event $e$ is not of type `TimeStepEvent`, $e$ is buffered. Each event is mapped to a time interval with the *filter* function ($t_e \mapsto [t_i, t_{i+1}[$) where $t_e \in [t_i, t_{i+1}[$ is the event timestamp and $t_i$ and $t_{i+1}$ are consecutive values of discrete time in the simulation. Hence for a given time $t$, the agent has to process a list of events $\{e | e \in \mathbb{E}, \mathit{filter}(t) = \mathit{filter}(t_e)\}$. If the event $e$ is not explicitly

timestamped, then the default timestamp is assumed to be equal to the time of the next simulation step (computed by the `nextTimeStep` function in Algorithm 1.3).

In a simulation agent, if the received event $e$ is of type `TimeStepEvent`, the current simulation *time* is updated with the timestamp of $e$. Due to our conservative synchronization approach, this timestamp is equal to the global time simulation. Additionally, the buffered events are consumed and fired into the current simulation agent by using the default event listener (provided by the execution platform). Finally, the `SynchronizationAwareSkill` implements the two functions of the `Behaviors` capacity that correspond to the two methods for receiving events: the `asEventListener` and `wake` functions.

```
skill SynchronizationAwareSkill implements Behaviors, Time {
  val defaultSkill : Behaviors
  val eventBuffer : EventListener
  var time : Integer
  new (platformSkill : Behaviors) {
    defaultSkill = platformSkill; eventBuffer = new InternalBuffer
  }
  def asEventListener : EventListener { return eventBuffer }
  def wake(e : Event) { eventBuffer.receiveEvent(e) }
  def getTime : Integer { return time }
  def nextTimeStep : Integer { return time + 1 }
  class InternalBuffer implements EventListener {
    val buffer : Map<Integer, Collection<Event>> = new MultiMap
    def receiveEvent(e : Event) {
      if (e instanceof TimeStepEvent) {
        time = e.timestamp
        var events = buffer.remove(time)
        for (be : events) {
          defaultSkill.asEventListener.receiveEvent(be)
        }
        defaultSkill.asEventListener.receiveEvent(e)
      } else {
        var timestamp = if (e instanceof TimestampedEvent) e.timestamp
                        else nextTimeStep
        if (timestamp > time) {
          buffer.put(timestamp, e)
        }
      }
    }
  }
}
```

**Algorithm 1.3.** Skill implementation of behaviors and time capacities.

The second built-in capacity that must be overridden to enable event synchronization is the `Time` capacity. This capacity provides the `getTime` function that is returning the current simulation time. In Algorithm 1.3, we define the local attribute `time`, which is the local simulation time from the agent point of view. According to our conservative approach, this local time is updated with the global simulation time that is the timestamp of a received `TimeStepEvent` occurrence.

In order to use the previously defined `SynchronizationAwareSkill` skill, it must be given to the simulation agent as the skill to be used when the functions of `Behaviors` and `Time` are invoked. In order to ensure that the synchronization process is hidden to the agents, we cannot change the definition of the simulation agents. Algorithm 1.4 describes this discarded approach, which is based on

the explicit creation of an instance of the SynchronizationAwareSkill skill, with the Behaviors skill from the platform as argument. This skill instance is mapped to the two capacities Behaviors and Time. From this point the agent is automatically synchronized with the rest of the system.

```
agent SimulationAgent {
  on Initialize {
    var syncSkill = new SynchronizationAwareSkill(getSkill(Behaviors))
    setSkill(syncSkill, Behaviors, Time)
  }
  [...]
}
```

**Algorithm 1.4.** Bad practice: explicit set of the synchronization skill in the simulation agents.

We consider that a better approach is to install the SynchronizationAwareSkill skill when a another simulation-based skill is installed into the agent. We have defined the EnvironmentInteractionCapacity capacity in Sect. 4.1. The corresponding skill may be defined in order to install the synchronization skill when it is installed, as illustrated by Algorithm 1.5.

```
skill SimulationEnvironmentInteractionSkill implements
    EnvironmentInteractionCapacity {
  def install {
    var syncSkill = new SynchronizationAwareSkill(getSkill(Behaviors))
    setSkill(syncSkill, Behaviors, Time)
  }
  [...]
}
```

**Algorithm 1.5.** Good practice: installing the synchronization skill from another simulation skill.

## 5 Performance

In order to be able to measure the performance without being biased by application related calculations, we created a very simple ping-pong application.

In the time period starting at $T_0$, every agent has 20 % probability to emit a *ping* message to $X$ other agents where $X \sim \mathit{Uniform}(1 : 100)$. The message needs to be delivered in the time period $T_d \sim \mathit{Uniform}(T_e - T_0)$ where $T_e$ denotes the end of simulated time. The measured time is illustrated in Fig. 1, where $T_0$ and $T_1$ denote the start and the end of the interval; $T_a$ denotes the end of the reception of the events sent by the environment to the simulation agents; $T_b$ the end of "application level payload work" done by the agents, and finally $T_c$ the end of delivering the AgentIsReadyEvent to the environment. For every time period, the amount of emitted messages is computed together with the total amount of time needed to execute this time period. Experiments are realized for 200 agents on a Linux Ubuntu 14.04LTS laptop with 8GB memory and a Intel Core i5-4210M CPU 2.60GHz × 4. The number of time periods that are simulated is 2 500.

Experimental results are illustrated in the graph represented in Fig. 2. In our experiments, all the agents have the same actions to do. Consequently, they

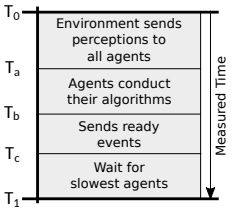


**Fig. 1.** An overview of the time measurement in our experiments.

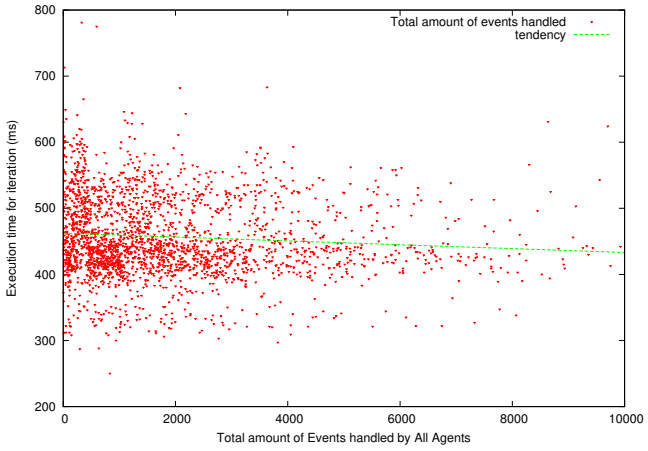


**Fig. 2.** Graph that represents the total amount of events handled in a specific iteration (x-axis) against the total execution time for that iteration in ms (y-axis) for the case of 200 agents and 2 500 iterations.

have approximately the same execution time. It is clear to see that the execution time follows a constant tendency, and hence seems to be independent of the number of processed events over the full range of observations. The execution time for a single period between two consecutive increments of simulated time includes: perception of the environment, application specific *payload* work and end-of-period notification. The duration required for the payload work in the experiment is negligible. The large variance of the execution time masks the expected dependency on the number of events.

## 6 Conclusion and Perspectives

A proof of concept is given for the support of the event synchronization using the SARL language and its Janus execution platform, without changing neither the specification of SARL nor the code of the Janus platform. Similar to Weyns and Holvoet [13], we plan refining our model by including regional synchronization. Another perspective is to provide an optimistic synchronization model. From a technological point-of-view, our synchronization mechanism will be included into the Janus execution platform.

**Acknowledgments:** the research reported was partially funded by the IWT 135026 Smart-PT: Smart Adaptive Public Transport (ERA-NET Transport III Flagship Call 2013 “Future Traveling”).

## Bibliography


[1] Braubach, L., Pokahr, A., Lamersdorf, W., h. Krempels, K., o. Woelk, P.: A generic simulation service for distributed multi-agent systems. In: In From Agent Theory to Agent Implementation (AT2AI'04. pp. 576–581 (2004)

[2] Fujimoto, R.: Parallel discrete event simulation. Communications of the ACM 33(10), 30–53 (1990)

[3] Fujimoto, R.: Parallel and Distributed Simulation Systems. Wiley, New York (2000)

[4] Galland, S., Gaud, N.: Organizational and holonic modelling of a simulated and synthetic spatial environment. E4MAS 2014 - 10 years later, LNCS 9068(1), 1–23 (Nov 2015), `http://www.springer.com/us/book/9783319238494`

[5] Jafer, S., Lui, Q., Wainer, G.: Synchronization methods in parallel and distributed discrete-event simulation. Simulation Modelling Practice and Theory 30, 54–73 (2013)

[6] Jefferson, D.: Virtual time. ACM Trans Program Lang Syst 7, 404–425 (1985)

[7] Lamport, L.: TI clocks, and the ordering of events in a distributed system. Communications of the ACM 21, 558–565 (1978)

[8] Michel, F.: The IRM4S model: the influence/reaction principle for multiagent based simulation. In: Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS07). ACM, Honolulu, Hawaii, USA (2007)

[9] Perumalla, K.: Parallel and distributed simulation: traditional techniques and recent advances. In: Proceedings of the 2006 Winter Simulation Conference. pp. 84–95. Monterey, CA (2006)

[10] Perumalla, K., Fujimoto, R.: Virtual time synchronization over unreliable network transport. In: Proceedings of the 15th International Workshop on Parallel and Distributed Simulation. pp. 129–136. Lake Arrowhead, CA (2001)

[11] Rodriguez, S., Gaud, N., Galland, S.: Sarl: A general-purpose agent-oriented programming language. In: Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2014 IEEE/WIC/ACM International Joint Conferences on. vol. 3, pp. 103–110 (Aug 2014)

[12] Tropper, C.: Parallel discrete-event simulation applications. Journal of Parallel and Distributed Computing 62(2), 327–335 (2002)

[13] Weyns, D., Holvoet, T.: Model for situated multi-agent systems with regional synchronization. In: Concurrent Engineering, Agents and Multi-agent Systems. pp. 177–188. Madeira, Portugal (2003)

[14] Weyns, D., Omicini, A., Odell, J.: Environment as a first-class abstraction in multi-agent systems. Autonomous Agents and Multi-Agent Systems 14(1), 5–30 (2007)

[15] Xu, Y., Cai, W., Aydt, H., Lees, M., Zehe, D.: An asynchronous synchronization strategy for parallel large-scale agent-based traffic simulations. In: SIGSIM-PADS'15. London, United Kingdom (Jun 2015)