mathematics

MDPI

Article

ActressMAS, a .NET Multi-Agent Framework Inspired by the Actor Model

Florin Leon

Faculty of Automatic Control and Computer Engineering, "Gheorghe Asachi" Technical University of Iasi, Bd. Mangeron 27, 700050 Iasi, Romania; florin.leon@academic.tuiasi.ro

Abstract: Multi-agent systems show great promise in the actual state of increasing interconnectedness and autonomy of computer systems. This paper presents a .NET multi-agent framework for experimenting with agents and building multi-agent simulations. Its main advantages are conceptual simplicity and ease of use, which make it suitable for teaching agent-based notions. Several algorithms, protocols and simulations using this framework are also presented.

Keywords: multi-agent framework; .NET framework; simulations; agent-based systems; agent algorithms; software design

1. Introduction

Currently computer systems are increasingly interconnected and the complexity of tasks that they solve requires less human intervention and an extended degree of autonomy. The promise of Internet of Things and autonomous cars and drones, including those aimed at delivering goods to customers, are prominent examples. There are also a large number of complex systems (e.g., social, economic, ecological) that can be studied using a bottom-up approach for modeling and simulation. These methods of analyzing the results of complex interactions are easier to apply than traditional, analytical models. Although multi-agent systems (MAS), arguably, still have to find some successful "show-off" applications, similar to the recent success of deep learning in artificial intelligence, they are an active area of research. Therefore, many MAS frameworks have been proposed with the goal of helping the user to focus on the high-level behavior rules and interaction protocols, rather than on the low-level details of concurrent and distributed programming.

The creation of ActressMAS [1,2] was fueled by two main reasons. First, it was based on the personal experience of the author in teaching multi-agent systems. Unfortunately, many popular frameworks require some effort of understanding a specific agent language, complex configurations of the platform itself, or some idiosyncrasies of the programming model. Therefore, the main goal of the proposed ActressMAS framework was simplicity. It requires minimal configuration, uses a mainstream programming language, and allows the user to focus on agent behavior, rather than learning the characteristics of the framework itself; therefore, it has proved successful in teaching MAS concepts. Secondly, it was found that while many multi-agent frameworks are based, e.g., on Java, not so many exist for the .NET ecosystem.

Thus, ActressMAS is a simple-to-use .NET multi-agent framework inspired by the actor model. This paper aims to present the philosophy of this framework, the main design decisions and the compromises that had to be made.

The rest of the article is organized as follows. In Section 2, some related work about other multi-agent frameworks available today is presented. Section 3 points out the similarities and differences between actors and agents. Section 4 describes the architecture of ActressMAS. Section 5 illustrates the performance of the framework on some benchmark problems and Section 6 contains a discussion about how ActressMAS relates to existing standards and to certain features found in other agent frameworks. The conclusions of this

check for updates

Citation: Leon, F. ActressMAS, a .NET Multi-Agent Framework Inspired by the Actor Model. *Mathematics* **2022**, *10*, 382. https://doi.org/10.3390/math10030382

Academic Editors: Ioannis G. Tsoulos and Jianquan Lu

Received: 5 November 2021
Accepted: 24 January 2022
Published: 26 January 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

work are presented in Section 7, while Appendix A presents several applications (i.e., agent algorithms and protocols, as well as multi-agent simulations) together with specific details about the examples included in the publicly available GitHub repository and the ways of using the proposed framework in other projects.

2. Related Work

Many applications of multi-agent systems have been put forward. They can be grouped into several categories, of which one can mention the following [3]:

- Social simulations: the study of population dynamics, the evolution of social corruption, or models of civil violence;
- Mobility simulations: traffic situations with the goal of analyzing traffic jams, adaptive traffic lights, route choice, mobility planning systems, urban planning based on accessibility studies with dynamic populations, microscopic models of pedestrian crowds and evacuation of buildings, or air traffic control;
- Physical entities: robots and autonomous vehicles (cars, drones) seen as agents;
- Environment and ecosystems: simulations in ecology, biology, climate models, human and nature interaction (sometimes using geographic information systems), epidemiology (the spread of infections or diseases);
- Organizational simulations: planning and scheduling, enterprise and organizational behavior, workflow simulations;
- Economic studies: business, marketing, economics, e.g., price forecasting in real-world markets;
- Medical applications: personalized healthcare or hospital management;
- Industrial simulations: manufacturing and production, including the use of holons;
- Military applications: military combat simulations, air defense scenarios;
- Distributed computing, e.g., in cloud computing, virtualized data centers, large-scale parallel or distributed computing clusters and high performance supercomputers;
- Games or movie-making.

Consequently, many agent frameworks have been proposed, with largely different levels of scope, performance and adoption. In the following, some examples are referenced, which seem representative of their corresponding categories. It must be emphasized that no established consensus exists towards the degree of relevance of specific platforms, so this is not necessarily a ranking in terms of popularity. However, the frameworks included below have a consistent user base, as well as associated research papers and projects. The main categories are those identified in [3], which is a comprehensive review of both active and inactive multi-agent frameworks.

The first category includes general-purpose platforms. One representative is JADE [4], a FIPA-compliant middleware made in Java, where agents are programmed in terms of "behaviors", a specific way to handle concurrent execution. Another is MASON [5], also a Java framework focused on discrete-event multi-agent simulation, with 2D and 3D visualization capabilities. One can also mention Orleans [6], which is one of the few agent frameworks available to .NET programmers. It uses virtual actors whose activations are performed in a turn-based asynchronous manner. The fundamental entity is a "grain", which has user-defined identity, behavior, and state.

From the category of platforms for cognitive and social studies, one can point out two examples of cognitive architectures. ACT-R [7] is a hybrid architecture with a symbolic component–a production system–and a subsymbolic one in the form of a set of massively parallel processes modeled with mathematical equations, which control many symbolic processes and are generally responsible for learning. Procedures can be expressed similarly to the brain's action selection mechanisms. Soar [8] is another cognitive architecture that aims to identify the building blocks necessary for an agent with artificial general intelligence, i.e., an agent able to perform many tasks in various domains. Soar is the final point of an evolution that started with the Logical Theorist [9], often considered "the first artificial intelligence program", designed to perform symbolic automated reasoning (proofs of

mathematical theorems) and shown during the Dartmouth workshop in 1956, the birthplace of the artificial intelligence field. One of the features of Soar is the use of "chunking" as a learning mechanism: once a (sub)goal is achieved, a rule or set of rules are added to the long-term memory expressed as a production system. Perhaps in the same category one can also include Jason [10], which implements a practical reasoning architecture (Belief-Desire-Intention, BDI [11]), using a special-purpose, logic-based programming language called AgentSpeak, the continuator of one of the first agent-oriented languages [12].

There are also platforms for artificial intelligence research, e.g., OpenAI Gym [13] that focuses on reinforcement learning environment simulations, control tasks, Atari games emulators that allow custom agents to play, and DeepMind Lab [14] that includes 3D navigation and puzzle-solving environments for intelligent agent experimentation, especially with deep reinforcement learning.

From the category of platforms for modeling and simulating natural and social phenomena, one can mention NetLogo [15]. It has a large library of "models", i.e., configurable multi-agent simulations with graphical user interface (GUI). Its main drawback is, arguably, its specific programming language based on Logo, which is quite different from other popular mainstream programming languages. Specific concepts are: "patches", i.e., cells in a grid similar to the cells in cellular automata, "turtles", i.e., agents that can move freely through space, and "links", which define connections between turtles and can be used to build network models.

Among the platforms for transport-related simulations, one can mention Carla [16], which focuses on autonomous driving systems and provides a physics engine for realistic 3D traffic scenarios simulations, and MATSim [17] for large-scale agent-based transport simulations.

Finally, a high-performance framework that uses the actor model is Akka [18], designed to build highly concurrent and distributed message-driven applications in Java and Scala. It can also be used with .NET languages by means of Akka.NET [19].

To summarize, all agent systems have certain commonalities, such as the need for support for parallel/distributed computations or communication, and the need to handle a reasonably large number of agents. The programmers should be able to focus on their specific tasks, not on these low-level details of the middleware, and this is the main goal of the various agent frameworks. However, they implement these requirements in very different ways, and integrate other concepts and ideas as well.

One can identify agent systems where the focus is on the autonomous behavior and on the interaction protocols, e.g., negotiation or coordination. These are supported by general purpose platforms, e.g., JADE. This somewhat contrasts with the multi-agent simulations, where many agents execute according to simple local rules, but the focus is on studying the complex interactions and the emergent behavior of the system. These are supported by platforms such as NetLogo. ActressMAS mainly belongs to the first type, but also allows the user to run simulations and build graphical user interfaces to observe the overall behavior of multi-agent systems.

3. Theoretical Aspects: Actors and Agents

Actors and agents are both entities that can be used for performing parallel and distributed computations. Both models rely on messages for exchanging information and do not recommend the use of shared memory structures. Still, there are conceptual differences between them that we point out as follows.

Actors can be described as computational processes that realize their functionality by sending and receiving messages to and from other actors in an asynchronous manner. These are the main operational axioms defined by the computational actor model [20]. In addition, an actor can create other actors. Actors are purely reactive entities because they can act only when they receive a message. In this computational model, the program flow is created by composing the individual behaviors of the actors in the system. They communicate only by sending messages and do not expose any part of their internal state to other actors. Each actor has an "inbox", i.e., a queue with the received messages, and processes them

in order, one at a time. Many actors usually exist in a system and they run concurrently. However, the model avoids the need for synchronization because each actor processes its own messages sequentially. For example, one can use a proxy actor for a shared resource. If two other actors need to access the resource, they can only send request messages to the proxy actor, and the proxy handles these messages one at a time; therefore, the resource cannot be directly accessed by two or more actors simultaneously.

On the other hand, agents can be defined as autonomous entities situated in their execution environment (i.e., they are embedded in their virtual/software or physical/hardware environment). In addition, intelligent agents should be capable of reactive, proactive and social behavior [21,22]. From this point of view, agents focus more on the capabilities that may allow them to be human representatives in various interaction scenarios, including those that may involve reasoning and planning.

The actor model can be seen as "weaker" than the agent model. While the autonomy and reactiveness can be ensured by the actor model, the proactive behavior is not so straightforward to model using pure actors. For example, an agent may need to act even if it receives no message from the outside. In order to be able to demonstrate complex social behavior, there should also exist a richer set of message semantics that the agents could use. Also, reasoning capabilities are not a requirement for actors, although such algorithms can be part of their internal logic.

While agents are autonomous entities that perceive their environment and can act upon it, actors can be controlled by other actors and may lack sensors and effectors [23].

Actors and agents are separate concepts, but the basic characteristics of actors such as parallel execution and reactive behavior triggered by message-based communication can constitute a starting point for the development of an agent system. In many cases, the behavior of agents is also driven by the messages they receive from other agents. However, in other cases agents should be able to take the initiative and act even when no messages are received, e.g., when something in their internal state changes and requires a certain action to be performed. In addition, since they are situated entities, they need some mechanism to access their environment. The latter requirement can also be accomplished by the exchange of messages between the environment and the agent, but when large amounts of data need to be transferred, this option may be less efficient.

An alternative direction of applications for agents is related to multi-agent simulations, which are concerned with building systems of interacting entities in a bottom-up manner, whose aggregated behavior can provide useful insights about the dynamics of complex systems. Many such simulations contain agents whose behavior is based on simple rules; therefore, the time needed to decide an action at a certain moment is more or less equal for all agents. In this situation, especially when there is a large number of agents involved, a turn-based execution is preferred, which gives each agent a chance to act, without the need of complex scheduling. Even parallel behavior may be emulated with a sequential execution of agents in random order during each turn.

Although initially inspired by the actor model, ActressMAS includes some mechanisms to handle proactiveness and to offer support for multi-agent simulations, thus departing from the pure actor model. Thais is why a turn-based approach was considered for agent execution. Beside its role in simulations, it can also be used to handle most protocols and algorithms by designing general agent programs with fine-grained decision logic. Still, it must be stated that this is not suitable when time-specific actions are needed to be performed by agents or for real-time applications. The turn-based execution also helps to implement the proactiveness feature of agents. The framework identifies the situation when no messages are received at the end of a turn, and this can be optionally used to change the agent state or to perform an action. For scenarios in which agents need to access certain parts of their environment (e.g., for indirect communication by stigmergy), ActressMAS also provides a shared memory structure present in the environment.

4. ActressMAS Architecture

In this section, the architecture of ActressMAS is presented, both high-level and detailed. The representations are Unified Modeling Language (UML) class diagrams [24].

Figure 1 displays the general architecture. The main concepts of a multi-agent system can be observed: the agent and the environment. In order to handle parallel and sequential execution in an efficient, transparent way, the agents are stored either in a concurrent or "normal" C# dictionary. Agents communicate by sending messages. The upper part of the diagram addresses the distributed capabilities of the framework using the client-server model. On each machine there is a container that includes a "runnable" environment. Containers communicate through a server by means of a special type of messages. When agents migrate between containers, their state (or a part of their state) is serialized and sent to the destination container where the agent is restarted. The architecture also contains a class used by the agents to filter the agents with certain properties in their environment and to "observe" them automatically in a perception method that can be used to update their beliefs before acting.

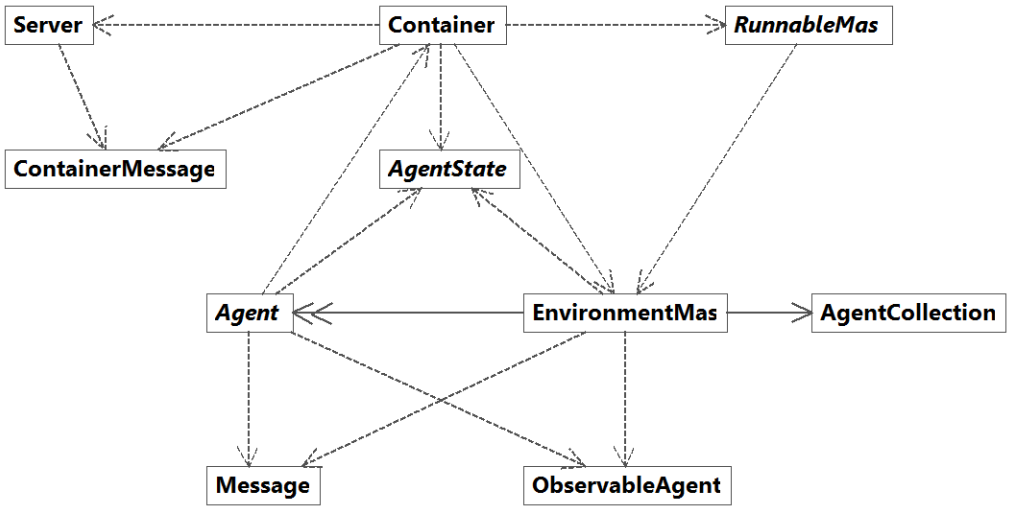

Figure 1. The general architecture of the ActressMAS framework.

All these classes will be detailed as follows.

4.1. Fundamental Features

Even if agents are the main way of expressing programming logic in an agent-oriented application, the description of ActressMAS will probably be clearer if we start with the description of the environment first. By definition, agents inhabit an environment of which they are a constituent part. They perceive it and act upon it. The entities involved in these operations may be other agents or properties of the environment.

In ActressMAS, the execution of the environment is based on turns. The maximum number of turns of a simulation is one of the parameters of the constructor of the environment class. The agents are not directly aware of this execution model, but the user can make this information available to the agents, as explained below. The turn-based execution was chosen in order to treat the two types of agent systems (the interaction of autonomous agents and multi-agent simulations) in the same way. The agents can be run sequentially or concurrently. In the latter case, the choice of turn-based execution (when the acting behavior of all agents is executed in parallel but all agents need to finish before starting the next turn) may be problematic if one agent takes much longer than the others to execute its behavior. However, there are at least two practical solutions for this situation. One solution is to place the long-running agent into a separate container which can also run on the same machine with the container that hosts the rest of the agents. The special agent can, e.g., perform intensive computations and report the results only at the end without

blocking the others. Another solution is to design the acting behavior of all agents in a fine-grained manner so that only atomic computations should be done at one time, while responding to messages.

Thus, the agents execute in a turn-based manner in both sequential and concurrent settings, but if their actions are properly designed, the parallel execution benefits from the multiple processor cores, when available, and the overall performance is faster.

The parallel execution is performed by launching a *Task* for the current behavior of each agent. Tasks are a lightweight form of implementing asynchronous behavior in .NET. They use a thread pool which is managed transparently by the .NET framework and allows the execution of a large number of agents. Using this mechanism, the user can create, e.g., tens of thousands of agents. Attempting to create a similar number of threads would likely block the operating system.

In case of sequential execution, the user can also choose that the agents run in the order in which they have been added to the environment, or in a random order. While in most cases the random order is natural, there are protocols in which the user can ensure that, e.g., a manager agent receives messages from all the worker agents in a turn before starting another round of the protocol. Placing the manager as the last agent simplifies the implementation, because otherwise the messages of some agents may only be received in the next turn and the manager would have to include a mechanism to count the number of messages received so far, or to identify the actual agents that have responded.

The UML class diagram of the environment class is presented in Figure 2. The name of the class is *EnvironmentMas* to avoid a conflict with the .NET *System.Environment* class. If these classes from both *ActressMas* and *System* namespaces were used together, the user would have to use a namespace-qualified name such as *ActressMas.Environment* in his/her code. But since the other ActressMAS classes do not have this necessity, it was decided that *EnvironmentMas* was a more appropriate name.

The environment includes the typical methods for adding, removing or enumerating agents. Except for agent creation, these operations are usually done using agent names, not references to the agent objects.

The environment also acts as a bidirectional proxy for sending messages between agents and moving agents between containers.

It also has some special methods that allow the programmer to handle turns in an explicit manner. For this purpose, the user must create a subclass of *EnvironmentMas* and override the *TurnFinished* and/or *SimulationFinished* methods. This is especially useful for multi-agent simulations, where the user can compute, e.g., some statistics after each turn or introduce external conditions or events at special moments in the simulation.

The environment also has a shared memory in the form of a dictionary where agents can record any kind of object with a string key. While in multi-agent systems communication is normally done by messages, from the practical point of view there are cases when having a shared memory greatly reduces the communication overhead, e.g., when agents need to be aware of some changing properties of the environment encoded as large objects. The shared memory facility should not be abused; however, it corresponds to the situation where physical agents perceive and manipulate objects in their environment.

The internal methods marked with a tilde are accessible to the other classes of the *ActressMas* assembly, but invisible to the user programs.

The environment class uses a special structure to store the agents, named *AgentCollection*, displayed in Figure 3. Since the agents have unique names, a dictionary is used to handle the agent objects. However, the collection of agents may change dynamically during the execution of the user program, e.g., when agents are added or removed during the execution of a turn. When the agents run in parallel and perform this kind of behavior, the access to a simple C# *Dictionary* must be used in conjunction with a lock, which can degrade performance. That is why a *ConcurrentDictionary* (a thread-safe collection that can be accessed by multiple threads concurrently) is used to store the agents when the environment is set for a parallel execution.

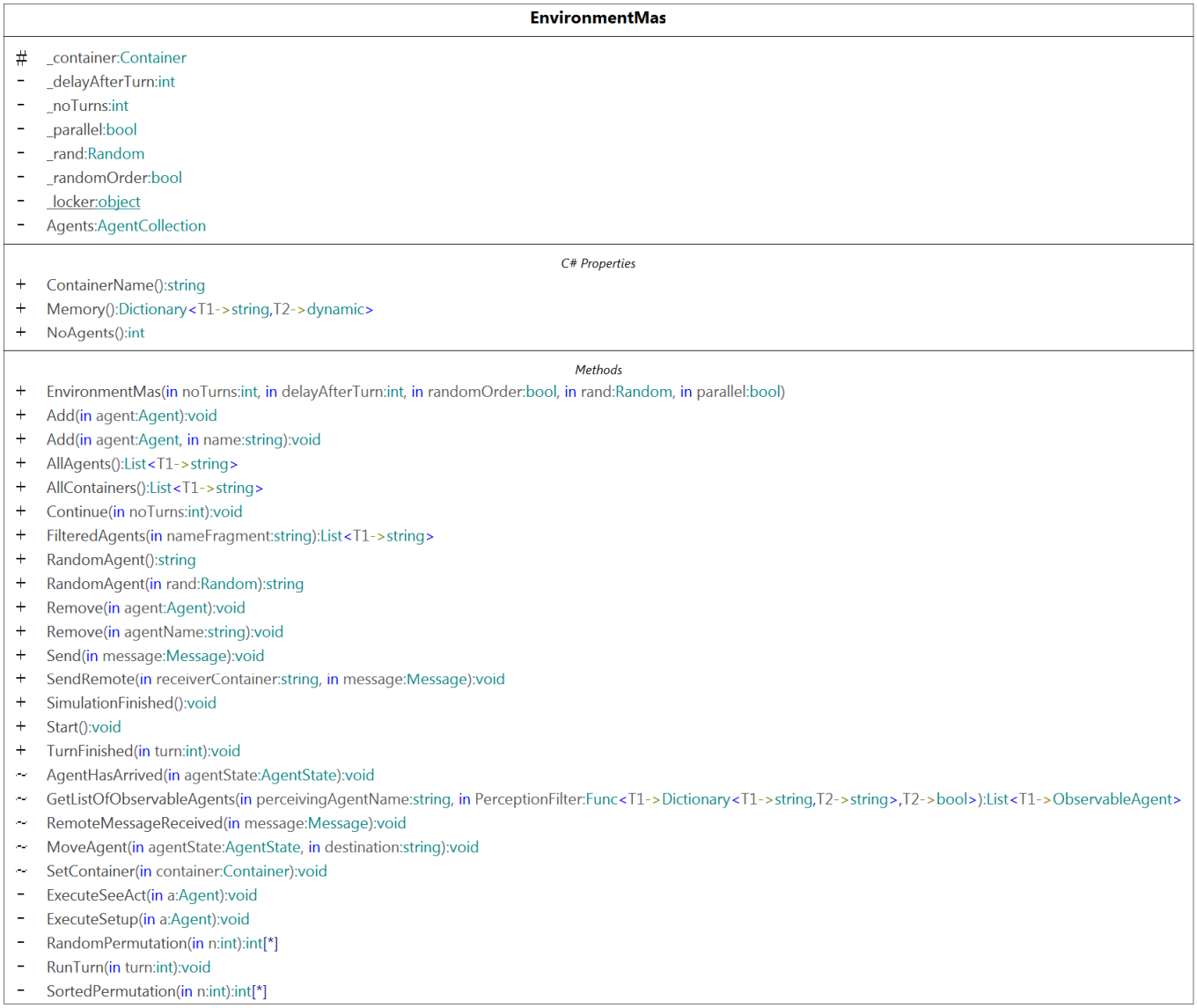

Figure 2. The environment class.

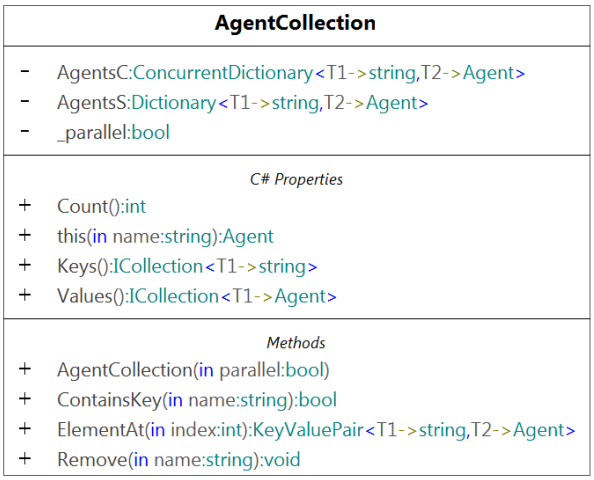

Figure 3. The agent collection class used by the environment to distinguish between concurrent and sequential behavior in a transparent way.

This class is designed as a kind of discriminated union. Currently (as of version 7), C# lacks support for this type of structure. The environment object uses either a *ConcurrentDictionary* or a "normal" *Dictionary*, depending on the *parallel* flag that shows whether the agents are run concurrently or sequentially. By using a dual collection, of which only one is allocated and actually used, an additional class hierarchy is avoided. All the public methods and properties are those common to dictionaries, therefore the *AgentCollection* object is used transparently as a generic dictionary, regardless of the underlying concurrent or sequential implementation.

Agents are the central entities of any program using the framework. The UML class diagram of the *Agent* class is presented in Figure 4. This is the base class for all the user-defined agents.

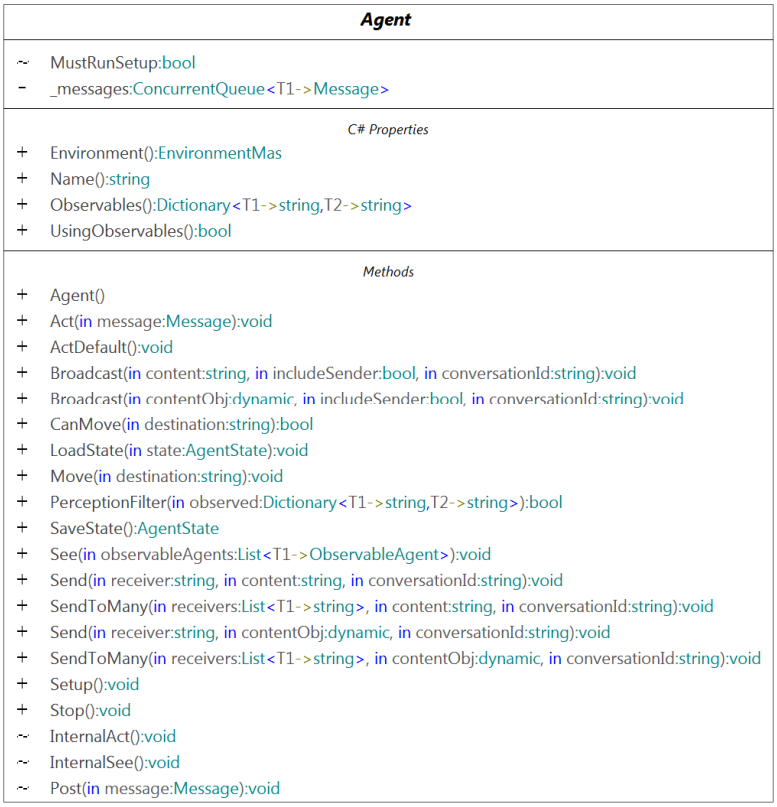

Figure 4. The abstract agent base class.

Each agent is registered in its environment by a unique name. This name is used in most operations, e.g., sending a message to another agent, which is the main form of communication in an agent-oriented program.

The typical methods that contain the logic of the agent are: *Setup*, *Act*, and *ActDefault*, which will be presented next. Although agents can execute in parallel, the code in these methods–for one agent–is always executed sequentially. This is one of the main strengths of the actor model, which avoids the need of synchronization for the access to critical resources, as explained in Section 1.

The *Setup* method is used for initialization. An agent class may also have a constructor, but as intended by the design of ActressMAS a constructor should be used to initialize the internal data structures such as lists, dictionaries, random number generators, etc. The *Setup* method should be used for agent-related logic, e.g., sending the initial messages that start the multi-agent protocol. Custom constructors may also be added to an agent class; a constructor with parameters can usually be employed when agents should be given some initial values for certain properties from the "outside" of the multi-agent system, i.e., when the agents are created and before the environment is started.

The *Act* method is activated when an agent receives a message. If there are more messages to be received, the *Act* method is activated once for each message. This is one of main tenets of actor-based programming, which is embedded in the ActressMAS framework. However, agents need not be purely reactive. They can maintain a state and they can update it after each message, and act based on their overall state, not only the current message. For example, a manager agent can know how many worker agents there are in the environment using the *FilteredAgents* method if the worker agents have a common part of their names, such as "worker*NN*-agent". Then the manager can decrease a counter for each message received from a worker. When the counter gets to zero, it knows that all workers have reported and may send them a new series of tasks.

The *ActDefault* method distinguishes ActressMAS from a purely actor-based model. In agent systems, there are situations when agents should act based on other conditions than responding to messages. For example, an English auction agent can designate the winner when no agent sends any more bids. This cannot be properly modeled within the pure actor paradigm because the acting condition is not the receipt of a message, but the lack of any message. Therefore, the *ActDefault* method was introduced to handle the cases when no messages have been received at the end of a turn. This increases the proactive capabilities of the agents, as their reactive capabilities are covered by the normal *Act* method. The agents also have the possibility to wait for a few turns–by counting the elapsed turns in *ActDefault*–and then act. In the initial version of ActressMAS, which did not contain the *ActDefault* method, a *Timer* object was used to send "wake up" messages that the agents could react to. However, a dedicated method within the framework seemed to be a much more elegant way to handle such issues.

The agents communicate directly by messages (detailed below). The main method used for this purpose is *Send*, where the receiver agent is designated by name. An agent can also send a message to all the other agents in the environment by using the *Broadcast* method.

An agent contains a *Stop* method, as well, which is called to deactivate the agent, which is thus removed from the multi-agent system. As it can be seen in Figure 2, the environment has a *Remove* method, which can be accessed by an agent through its *this.Environment* property. As intended by the design of ActressMAS, the *Stop* method should be used when the decision to be stopped belongs to the agent itself, while the *Environment.Remove* method should be used when the decision to stop an agent belongs to some other agent or to an external factor. The latter case is not common in the autonomous agent protocols, but it is encountered in multi-agent simulations, e.g., in a predator-prey simulation, a predator can "kill" a prey. As one can notice, there is no *Start* method in the *Agent* class. Agents start automatically when the environment starts, or when new agents are added later to a running environment. If the user wants the agent to suspend its execution for several turns, this can be easily accomplished, e.g., by using a Boolean flag in the agent class and conditioning any acting behavior on its value.

There are a few other methods implemented in an agent class, but they will be described in the following subsections, which present the observable properties and the distributed capabilities of ActressMAS.

Messages are the only direct method for inter-agent communication; therefore, the class corresponding to a message (Figure 5) deserves special attention. It has been designed with loose inspiration from the FIPA ACL specification (the Agent Communication Language proposed by the Foundation for Intelligent Physical Agents) [25] and the implementation of messages in the JADE framework [4]. However, the goal was to allow the use of ACL concepts while maintaining a very simple syntax. Therefore, a message has a compulsory sender and receiver, and an optional conversation identifier, which can be used in some protocols to identify an ongoing sequence of messages that form a unique conversation. The sender is automatically assigned when an agent sends a message. Usually, only the receiver and the content must be specified.

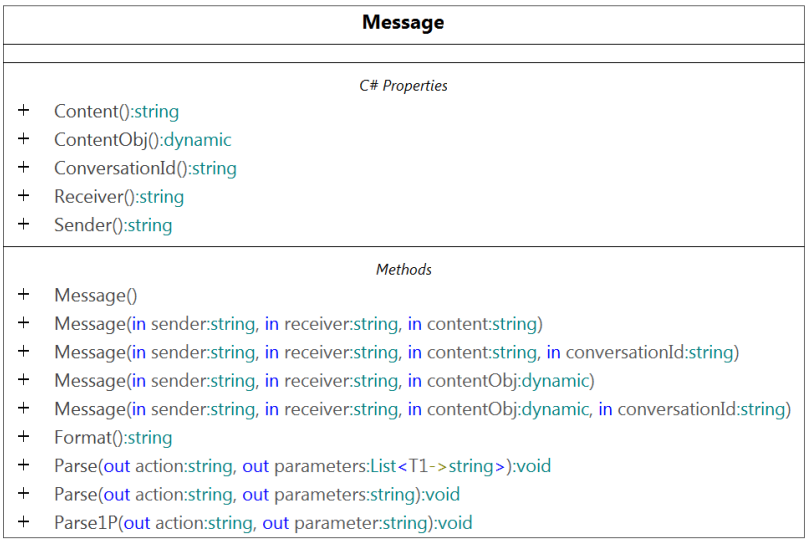

Figure 5. The class corresponding to the messages passed between agents.

The content can be expressed in two ways. The first is in the form of a string. The space-delimited words express the meaning. Typically, the first word defines the main action or message type, somewhat similar to an ACL performative, which shows the type of the communicative act, as inspired from the speech acts theory [26]. However, in ActressMAS there are no constraints about the values of this message type, i.e., the user can choose any name. The rest of the words specify the parameters. For example, if an agent wishes to send another agent its position on a two-dimensional grid, the content of the message can be "position 2 5". Most of the time, an agent that receives a message needs to split the string to be able to interpret the content. For this goal, the *Message* class contains several *Parse* helper methods, which identify the first word, called the "action" and the rest, called the "parameters". Depending on the protocol defined by the user, the parameters can be identified as a list of strings, one for each parameter, or a single string with all the parameters concatenated. When the action has only one parameter, the *Parse1P* method can be used.

Using strings for the content of the messages is very flexible and in line with the philosophy of agent communication, but can be less efficient because of the need to split the whole string into parameters and convert them to their specific types, e.g., integers or double-precision real numbers. Therefore, the second way of encoding the content is directly as objects. The user should define one or more custom classes and assign their instances to the *ContentObj* property of the message. The receiving agent can directly cast this property to the corresponding object type.

Finally, the *Message* class also contains a *Format* method, which can be used to display the pretty-printed message, with its sender, receiver and content.

If the receiver agent is in another container (a situation that uses the distributed infrastructure described below in Section 4.3), the receiver name should be qualified with the name of the container, e.g., "agent1@container2".

The basic way to create a multi-agent system is illustrated in the code below. First, the environment is created, then the agents are created and added to the environment, and finally the environment is started. When an agent is added to the environment, it is customary to assign it a name, because several agents from the same class can exist in the environment.

```
public class Program
{
    private static void Main(string[] args)
    {
        var env = new EnvironmentMas();
        var a1 = new Agent1(); env.Add(a1, "a1");
        var a2 = new Agent2(); env.Add(a2, "a2");
        env.Start();
    }
}
```

In this example, the *Setup* method simply displays a message and introduces a short delay, which is needed to emphasize the concurrent behavior of the agents. A single message is sent during the setup by each agent, and the *Act* method responds to the message received from the peer. If several messages had been sent, the *Act* method would have been activated for each of them. The *ActDefault* method is used when, at the end of a turn, no messages have been received. When this happens, the agent stops.

When there are no more agents in the environment, the simulation stops, even before the maximum number of turns specified by the user is reached.

```
public class Agent1: Agent
{
    public override void Setup()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"Setup: {i + 1} from a1*");
            Thread.Sleep(100);
        }

        Send("a2", "msg");
    }

    public override void Act(Message message)
    {
        for (int i = 0; i < 3; i++)
  {
            Console.WriteLine($"Act: {i + 1} from a1*");
            Thread.Sleep(100);
        }
    }

            public override void ActDefault()
    {
  Console.WriteLine("ActDefault: no messages for a1*");
        Stop();
    }
}
```

A similar structure is used by the second agent, but the number of messages in *Setup* and *Act* are different (3 instead of 10 and vice versa), in order to break symmetry.

```
public class Agent2: Agent
{
    public override void Setup()
    {
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine($"Setup: {i + 1} from a2");
            Thread.Sleep(100);
        }

        Send("a1", "msg");
    }

    public override void Act(Message message)
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"Act: {i + 1} from a2");
            Thread.Sleep(100);
        }
    }

    public override void ActDefault()
    {
        Console.WriteLine("ActDefault: no messages for a2");
        Stop();
    }
}
```

The output of this simple program can be seen in Figure 6. The asterisk is used to mark the reports of the first agent in order to help the reader to distinguish the behavior of the two agents more easily.

Setup: 1 from a2	Setup: 1 from a2
Setup: 1 from a1*	Setup: 2 from a2
Setup: 2 from a1*	Setup: 3 from a2
Setup: 2 from a2	Setup: 1 from a1*
Setup: 3 from a1*	Setup: 2 from a1*
Setup: 3 from a2	Setup: 3 from a1*
Setup: 4 from a1*	Setup: 4 from a1*
Setup: 5 from a1*	Setup: 5 from a1*
Setup: 6 from a1*	Setup: 6 from a1*
Setup: 7 from a1*	Setup: 7 from a1*
Setup: 8 from a1*	Setup: 8 from a1*
Setup: 9 from a1*	Setup: 9 from a1*
Setup: 10 from a1*	Setup: 10 from a1*
Act: 1 from a2	Act: 1 from a1*
Act: 1 from a1*	Act: 2 from a1*
Act: 2 from a1*	Act: 3 from a1*
Act: 2 from a2	Act: 1 from a2
Act: 3 from a1*	Act: 2 from a2
Act: 3 from a2	Act: 3 from a2
Act: 4 from a2	Act: 4 from a2
Act: 5 from a2	Act: 5 from a2
Act: 6 from a2	Act: 6 from a2
Act: 7 from a2	Act: 7 from a2
Act: 8 from a2	Act: 8 from a2
Act: 9 from a2	Act: 9 from a2
Act: 10 from a2	Act: 10 from a2
ActDefault: no messages for a2	ActDefault: no messages for a1*
ActDefault: no messages for a1*	ActDefault: no messages for a2

Figure 6. The output of the simple program used to exemplify the basic agent methods of ActressMAS. The results are displayed when agent behaviors are executed: left: in parallel; right: sequentially but in a random order.

Figure 6(right) shows the results when the environment is set to execute agents sequentially by using:

```
var env = new EnvironmentMas(parallel: false);
```

The agents can be executed sequentially and in the order in which they have been added to the environment by using:

```
var env = new EnvironmentMas(randomOrder:false, parallel: false);
```

In this case, agent *a1* would be the first to report in all three methods: *Setup*, *Act*, and *ActDefault*.

4.2. Observable Properties

Especially in multi-agent simulations, the next state of an agent may depend on the current state of its neighbors, e.g., simulations related to cellular automata. Perhaps agents may have a limited field of view and may perceive only a subset of the agents in their environment. In order to facilitate the handling of such situations, ActressMAS includes observable properties. They are implemented as a dictionary where the key is a string, i.e., the name of the property, and the value can be any kind of object. At initialization, e.g., in the *Setup* method, the agents define these properties. Then the agents override the *PerceptionFilter* predicate which defines the conditions that make other agents visible or observable. For example, if an agent can only see its neighbors within a certain radius, the predicate should express the condition that the Euclidian distance between the position of the "ego" agent (the *this* object) and the position of a neighbor agent (the *observed* parameter) be less than the specified radius. Initially, the agents should have defined an observable position property. Then the agent should implement the *See* method, called before *Act* or *ActDefault*, which provides the list of *ObservableAgent* objects (Figure 7) as a parameter. The agent can, for example, process or store this information in order to use it in the acting methods.

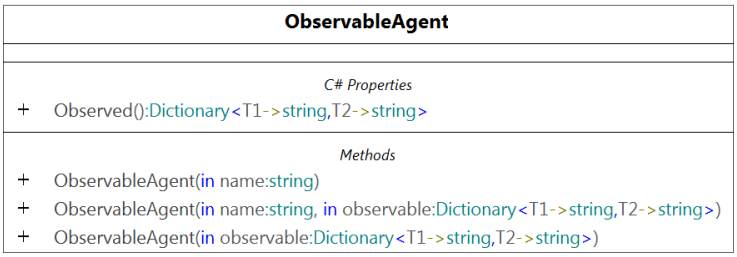

Figure 7. The class corresponding to observable agents.

The following code illustrates an example of using observables. The multi-agent system is defined in a similar fashion as in the previous example, but here the agents need to have the *UsingObservables* property explicitly set to be true.

```
public class Program
{
    public static void Main(string[] args)
    {
        var env = new EnvironmentMas(noTurns: 10, randomOrder: false, parallel: false);
        var a1 = new MyAgent(); a1.UsingObservables = true; env.Add(a1, "Agent1");
        var a2 = new MyAgent(); a2.UsingObservables = true; env.Add(a2, "Agent2");
        var a3 = new MyAgent(); a3.UsingObservables = true; env.Add(a3, "Agent3");
        env.Start();
    }
}
```

All three agents are instances of the same class, *MyAgent*. One can distinguish the *Perception Filter* method used to define the conditions that make a neighbor agent "visible" and the *See* method that provides the list of the agents that are observed in the current turn, before acting. Basically, each agent is assigned a random number between 0 and 30, and can only see the agents with similar numbers, i.e., when the difference between their corresponding numbers is less than 10. Since no messages are sent in this example, the main logic of the agents relies on the implementation of the *ActDefault* method.

```
public class MyAgent: Agent
{
    private List<ObservableAgent> _observableAgents = null;

    public override void Setup()
    {
        Observables["Name"] = Name;
        Observables["Number"] = $"{Numbers.GenerateNumber():F2}";
    }

    public override bool PerceptionFilter(Dictionary<string, string> observed)
    {
        double myNumber = Convert.ToDouble(Observables["Number"]);
        double obsNumber = Convert.ToDouble(observed["Number"]);
        return (Math.Abs(myNumber − obsNumber) < 10);
    }

    public override void See(List<ObservableAgent> observableAgents)
    {
        _observableAgents = observableAgents;
    }

    public override void ActDefault()
    {
        Console.Write($"I am {Name}. ");

        if (_observableAgents == null || _observableAgents.Count == 0)
            Console.WriteLine("I didn't see anything interesting.");
        else
        {
            Console.WriteLine($"My number is {Observables["Number"]} and I saw:");
            foreach (var oa in _observableAgents)
                Console.WriteLine(
                    $"{oa.Observed["Name"]} with number = {oa.Observed["Number"]}");
        }

        Observables["Number"] = $"{Numbers.GenerateNumber():F2}";

        Console.WriteLine($"My number is now {Observables["Number"]}");
        Console.WriteLine("————————————————————————————————————————");
    }
}
```

The class that generates random numbers for the agents is also presented below.

```
public class Numbers
{
    private static Random _rand = new Random();
    public static double GenerateNumber() => _rand.NextDouble() * 30;
}
```

The output of this program is presented in Figure 8.

```
I am Agent1. My number is 5.85 and I saw:
Agent2 with number = 9.48
My number is now 22.72
---------------------------------------------------------
I am Agent2. My number is 9.48 and I saw:
Agent3 with number = 17.06
My number is now 7.88
---------------------------------------------------------
I am Agent3. My number is 17.06 and I saw:
Agent1 with number = 22.72
Agent2 with number = 7.88
My number is now 7.82
---------------------------------------------------------
I am Agent1. I didn't see anything interesting.
My number is now 18.20
---------------------------------------------------------
I am Agent2. My number is 7.88 and I saw:
Agent3 with number = 7.82
My number is now 22.65
---------------------------------------------------------
I am Agent3. I didn't see anything interesting.
My number is now 8.71
```

Figure 8. The output of the program with observable agents.

4.3. Mobile Agents

Beside the ability to run agents concurrently, ActressMAS also supports mobile agents, which can stop their execution on one machine and resume their execution on a different one. In the following paragraphs, the distributed part of the architecture is presented, together with examples of using this capability.

The host of an environment on each machine is called a "container". This idea is inspired from the JADE framework [4]. However, in ActressMAS there is no distinction between a "main" container and "secondary" containers. Moreover, if the user does not intend to work with mobile agents, he/she does not need to define any container at all. Containers can be placed on different machines, but it is also possible to have multiple containers on the same machine.

Containers communicate by means of a server (Figure 9), which mainly keeps track of the active containers and passes messages between them.

The user must instantiate this class and may optionally define an event handler where the messages from the server can be accessed. In the example below, only the active containers are displayed.

A container (Figure 10) can be seen as a kind of proxy between an environment and the server. It manages the communication with the server (registers, deregisters and keeps a list of alive containers, received from the server). A container handles two main functions. First, when an agent wants to move, the container serializes the agent (actually, the desired part of its state, as explained below) and sends a corresponding message to the server, including the serialized state. The server routes the message to the destination container. There, the container deserializes the agent and informs the environment that an agent has arrived. Secondly, it routes remote messages between agents, i.e., from a container to another.

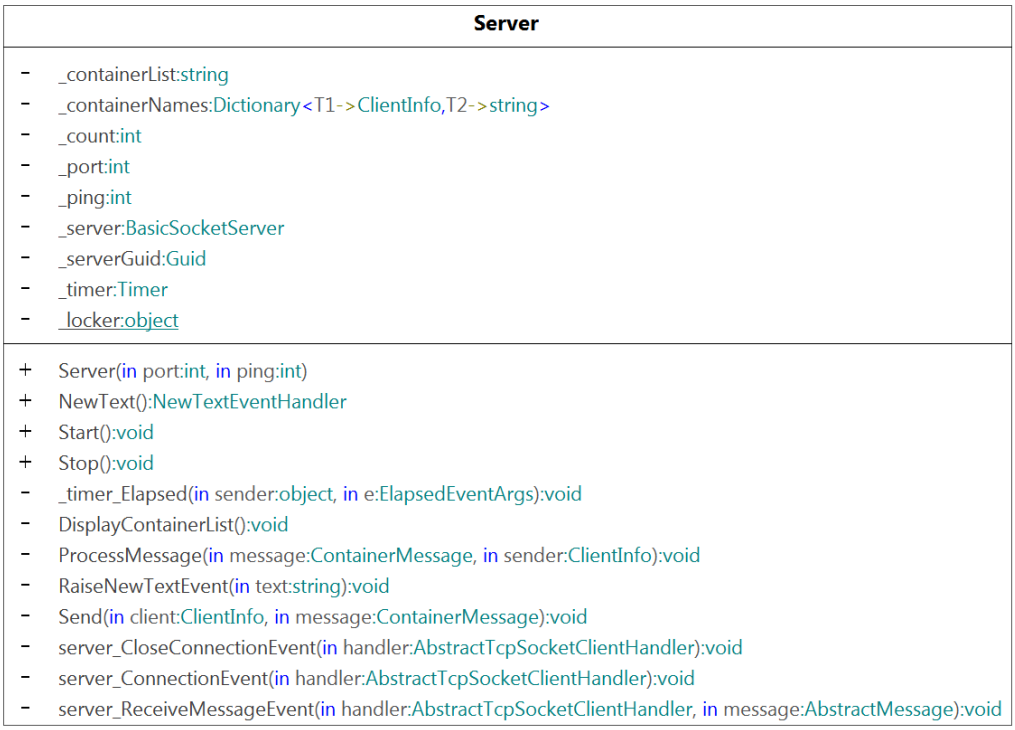

Figure 9. The server class.

```
public static class Program
{
    private static void Main()
    {
        var server = new Server(5000, 3000);
        Console.WriteLine("Server listening on port 5000.");
        server.NewText += server_NewText;
        server.Start();
        Console.WriteLine("Press ENTER to close the server.");
        Console.ReadLine();
        server.Stop();
    }

    private static void server_NewText(object source, NewTextEventArgs e)
    {
        if (e.Text.StartsWith("Containers:"))
        {
            Console.Clear();
            Console.WriteLine(e.Text);
        }
    }
}
```

The typical way of using the distributed capabilities of ActressMAS is first to initialize and connect the containers to the server. Then, the multi-agent environment in each container needs to be started. This is achieved by means of a simple class called *RunnableMas* (Figure 11), whose utilization is exemplified below.

The communication between a container and the server is done using a special type of message called *ContainerMessage* (Figure 12). Its structure is somewhat similar to an agent *Message*, but it includes the actual serialization and deserialization functionality, together with a special property, *Info*, which is used for handling the semantics of the message, e.g., "Request Register" (when a container wants to register to the server), "Inform Invalid

Name" (if the container name cannot be accepted by the server), "Inform Containers" (when the *Content* is a list with all available containers), "Request Move Agent" (when an agent has arrived), "Send Remote Message" (when an agent message is received in the *Content* property), etc.

The following paragraphs describe an example about creating a container with an environment that runs the agents. It includes the GUI in Figure 13, where one can see the relevant functionality: creating a container and starting it (connecting to the server), running the multi-agent system, and then stopping it (disconnecting from the server). For increased clarity and brevity, only the important parts of the methods are included, e.g., exception handling and the reading or writing of properties of GUI controls are omitted.

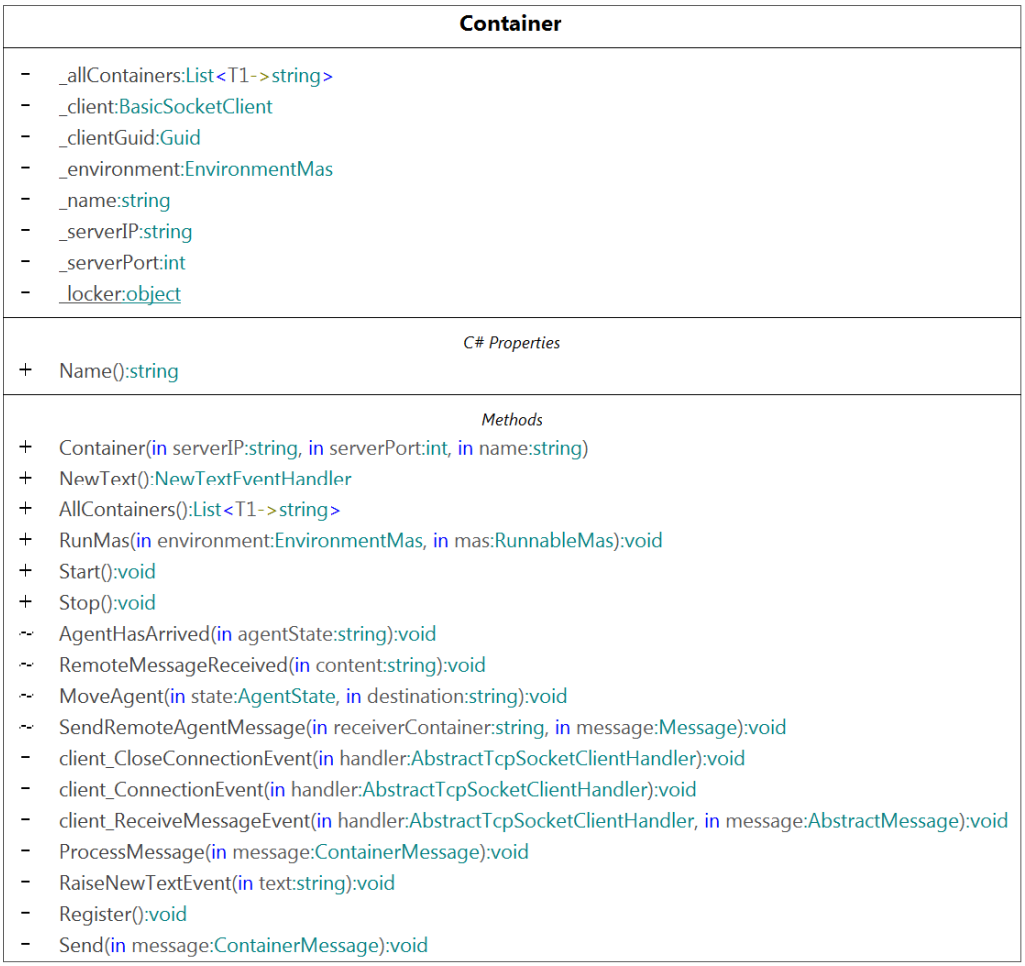

Figure 10. The container class.

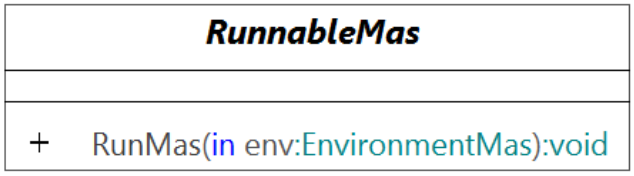

Figure 11. The class used to start running a multi-agent system in a container for distributed scenarios.

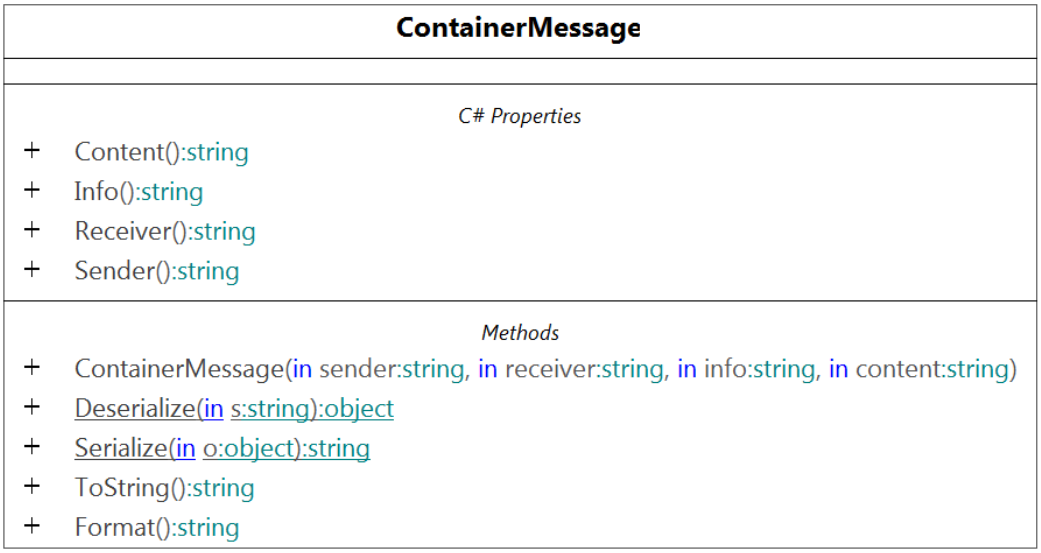

Figure 12. The class corresponding to the messages passed between containers and the server.

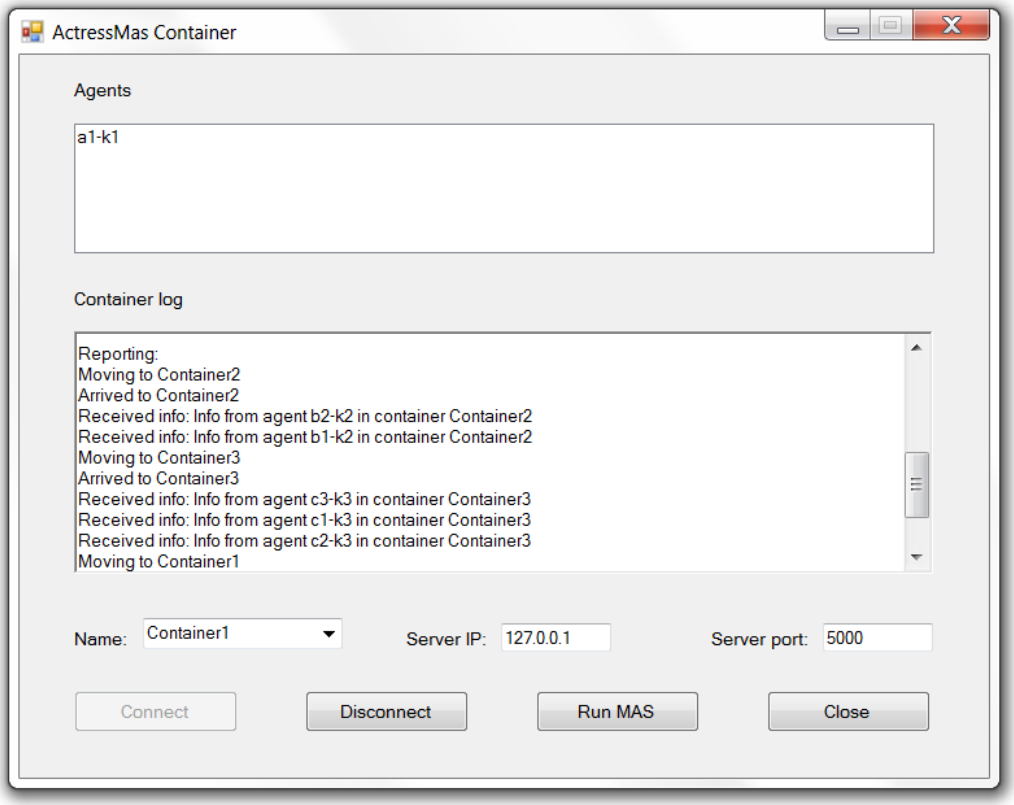

Figure 13. The graphical user interface of a program showing a multi-agent system with mobile agents.

```
private void buttonStart_Click(object sender, EventArgs e)
{
    _container = new Container(_serverIP, Convert.ToInt32(_serverPort), _containerName);
    _container.Start();
}

private void buttonRunMas_Click(object sender, EventArgs e)
{
    _environment = new EnvironmentMas();
    _container.RunMas(_environment, new MasSetup());
}

private void buttonDisconnect_Click(object sender, EventArgs e)
{
    if (_container != null)
        _container.Stop();
}
```

The connection between a container and an environment is made by means of a class derived from the *RunnableMas* class. In this way, the user can make a single application which can be run on different machines or containers, and the setup of the multi-agent system will be different depending on the specific container.

The example in the code below considers three containers with static agents, which just provide some piece of information when being asked, and a mobile agent, starting in "Container1" and then moving to the other containers. It gathers information from the static agents along the way and then returns home to "Container1" and reports the aggregated information.

```
public class MasSetup: RunnableMas
{
    public override void RunMas(EnvironmentMas env)
    {
        string home = env.ContainerName;

        switch (home)
        {
            case "Container1":
                // create a mobile agent and a static agent and add them to the environment env
                break;
            case "Container2":
                // create two static agents and add them to the environment
                break;
            case "Container3":
                // create three static agents and add them to the environment
                break;
        }

        env.Start();
    }
}
```

The next step is to create the agents, in the same way as described in Section 4.1. The following code specifies the mobile agent.

```
public class MobileAgent: Agent
{
    private string _log; // stores the pieces of information received from the static agents
    private Queue<string> _moves; // the path to follow when moving between containers
    private bool _firstStart = true;
    private int _turnsToWaitForInfo;

    public override void Setup()
    {
        if (_firstStart) // Setup is also called when arriving to a new container
        {
            _firstStart = false;
            _moves = new Queue<string>();
            foreach (string cn in Environment.AllContainers())
                if (cn != Environment.ContainerName) // home
                    _moves.Enqueue(cn);
            // return home, get local info and report
            _moves.Enqueue(Environment.ContainerName);
        }
        else
        {
            // the agent has moved to Environment.ContainerName
            Broadcast("request-info");
            _turnsToWaitForInfo = 3;
        }
    }

    public override void Act(Message message)
    {
        _log += $"Received info: {message.Content}\r\n"; // info from static agents
    }

    public override void ActDefault()
    {
        if (_turnsToWaitForInfo-- > 0)
            return;

        if (_moves.Count > 0)
        {
            string nextDestination = _moves.Dequeue();
            // checks whether the destination container is still active
            if (CanMove(nextDestination))
            {
                _log += $"Moving to {nextDestination}\r\n";
                Move(nextDestination);
                return;
            }
        }

        _log += "Stopping\r\n";
        Stop();
    }
}
```

A static agent has a much simpler logic.

```
public class StaticAgent: Agent
{
    private string _info; // the piece of information it reports

    public override void Setup()
    {
        _info = $"Info from agent {Name} in container {Environment.ContainerName}";
    }

    public override void Act(Message message)
    {
        if (message.Content == "request-info")
            Send(message.Sender, _info);
    }
}
```

When an agent is supposed to move to a different container, its state is serialized and sent by means of a container message to the destination container. There, a new object is instantiated, its state is set and it is added to the new environment. Thus, ActressMAS employs the concept of weak mobility [27], i.e., the value of the internal fields are preserved during the move, but the execution flow is not preserved: the agent has to finish a method (e.g., *Act*) on the source container, and start from another method (e.g., *Setup*) on its arrival at the destination; it cannot move in the middle of the execution of a method and resume from the next instruction at the destination.

The framework does not impose that the user marks the entire agent class as serializable, in order not to add any constraints to the agent implementations, especially since the user may not want to use mobile agents at all. For example, if the user needs a *Timer* object in an agent to define a recurrent event, that class is no longer serializable. Therefore, the user can choose the specific state that he/she wishes to be transferred when an agent moves. This is achieved by subclassing the *AgentState* class (Figure 14). The derived class must be serializable. This process uses the Memento design pattern [28] to save and restore the internal state of an agent. In this way, the user is also able to send only the relevant parts of the agent state. However, if the agent class itself is serializable, the whole state of the agent can be sent as a specific agent object.

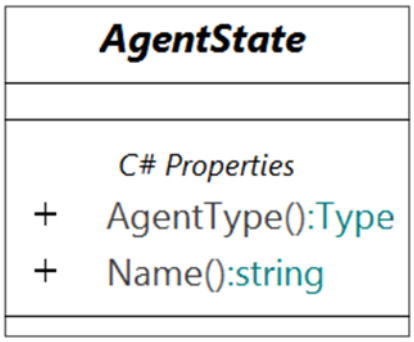

Figure 14. The class of the agent state used in conjunction with the Memento design pattern to ensure the movement of agents between containers in a transparent way.

For the mobile agent presented above, the following two methods can be added.

```
public class MobileAgent: Agent
{
    ...

    public override AgentState SaveState()
    {
        return new MobileAgentState
        {
            FirstStart = _firstStart,
            Log = _log,
            Moves = _moves
};
    }

    public override void LoadState(AgentState state)
    {
        var st = (MobileAgentState)state;
        _firstStart = st.FirstStart;
        _log = st.Log;
        _moves = st.Moves;
    }
}


[Serializable]
public class MobileAgentState: AgentState
{
    public bool FirstStart;
    public string Log;
    public Queue<string> Moves;
}
```

5. Performance for Some Benchmark Problems

This section attempts to give the reader an idea about the capabilities of ActressMAS with the help of two benchmark problems. They are typically used for actor-based reactive systems; therefore, a direct comparison with other frameworks is not completely objective, because ActressMAS has not been optimized for message passing or agent creation and is not only actor-based. Still, these benchmarks can provide indicative information about the speed of the framework, and this can help potential users to decide whether ActressMAS is appropriate for their particular needs.

The first problem is "Ping Pong". Each agent sends an initial message to all the other agents in the multi-agent system. Then, when an agent receives a message, it replies to the sender. This continues until a maximum number of messages is reached. In our case, scenarios with 10 agents and 10 million messages were considered. A slightly different version of this problem also exists, where a pair of agents exchange only a specified number of messages (e.g., 100), but this was not addressed in our experiments.

The second problem is "Skynet", used to measure the performance of actor creation and basic calculations. Each agent creates 10 children, each child agent creates another 10 children and so on, until a maximum number of agents is eventually reached. Each agent is also assigned a number, incrementally. On the final level, the agents send their ordinal numbers to their parent. Each parent sums these numbers and transmits the sum to its own parent and so on, until the initial root agent sums all the partial sums and reports the final sum.

The results obtained for different scenarios are presented in Table 1. The last column shows the average values out of 10 runs for each configuration. They were obtained using a computer with a 4-core 2 GHz Intel processor and 8 GB of RAM.

Table 1. Performance of the ActressMAS framework for two benchmark problems with various settings.

Benchmark/ Common Settings	Scenario	Results
Ping Pong 10 agents, 10,000,000 messages	parallel execution	1.516 s 6,462,079 messages/s
	sequential execution	3.880 s 2,577,297 messages/s
Skynet 10 children	10,000 agents parallel execution	3.800 s
	20,000 agents parallel execution	12.221 s
	50,000 agents parallel execution	95.611 s
	100,000 agents parallel execution	443.956 s
	10,000 agents sequential execution	0.600 s
	20,000 agents sequential execution	1.523 s
	50,000 agents sequential execution	13.620 s
	100,000 agents sequential execution	92.017 s

The volume testing capacity of ActressMAS seems lower than other professional actor frameworks; e.g., for Akka [18] and Proto.Actor [29] there are reports [29] of approximately 40 million and 120 million messages per second for Ping Pong, and approximately 4 s and 1 s for Skynet with 1 million actors, respectively, although their benchmark results are obtained for different hardware configurations. Therefore, ActressMAS should be used for applications where high performance such as very fast execution speed or a very large number of actors are not critical requirements. It may not be recommended for problems based on very large numbers of simple reactive actors.

As the agent creation benchmark shows, the internal data structures used to store and access agents may be further optimized beyond what the .NET framework offers. Less crucially, the implementation of the queue used by the agents to receive messages may also be improved to increase the number of processed messages.

However, ActressMAS is not only an actor-based framework, but has an especially constructed infrastructure for multi-agent systems, where agents can communicate using messages with custom structure and are not restricted to purely reactive behaviors. Moreover, it is open-source software, and a deliberate implementation choice was to prefer clarity over optimization.

6. Discussion

6.1. Relationship with FIPA Standards

FIPA specifications represent a collection of standards which are intended to promote the interoperation of heterogeneous agents and the services that they can represent. They try to define multiple aspects of agent systems. In this section, we refer to the FIPA specification for an agent abstract architecture [30] that specifies the necessary components of a so-called "Agent Platform", presented in Figure 15. An agent must be registered on a platform in order to interact with other agents.

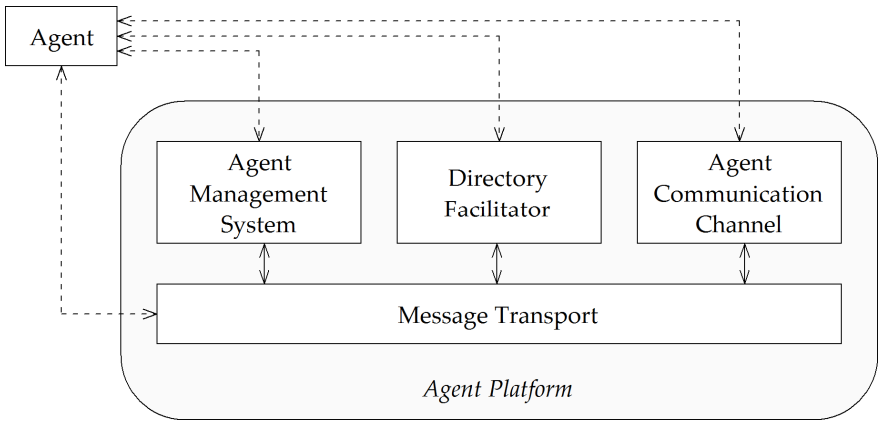

Figure 15. The components of the FIPA Agent Platform (adapted from [31]).

The FIPA specifications were initially intended to be used by various agent-based commercial systems that would need to address the issue of agent interoperability. However, over time such systems failed to materialize. Although more than one hundred agent frameworks have been created (some of them no longer under development) [3], not many had the compliance with the FIPA standards among their design objectives. JADE [4] is a prominent example of a FIPA-compliant agent framework, but a few others are FIPA-compliant as well.

Still, the abstract architecture proposed by FIPA deserves attention from a conceptual point of view. In the following, we discuss how ActressMAS relates to these four components, although it is not FIPA-compliant.

The Agent Management System (AMS) provides "white pages" services and life-cycle management, e.g., creation, deletion and migration of agents. Although this system is not explicitly defined as a separate entity, the environment in ActressMAS fulfils the role of the AMS because it stores the agents and is in charge of the operations mentioned above.

The Directory Facilitator (DF) provides "yellow pages" services for agents. In this case, the agents are seen as service providers and consumers. Some agents can register or deregister their services while others can look for specific services and attempt to use them. ActressMAS does not include a DF infrastructure, but the "yellow pages" functionality is implemented in one of the examples. A service broker agent maintains a collection of the service providers and the services they offer. The clients can interrogate it and receive the list of agents providing a certain service. Then, the clients (the service consumers) and the service providers can communicate directly.

The Agent Communication Channel is responsible for routing the messages to the agents located both on the current platform and on other platforms. The Message Transport service forwards the messages to the destination agents in order and is also responsible for the mapping between the logical names of the agents and their physical transport addresses. The messages are supposed to observe the Agent Communication Language (ACL) format, one of the major contributions of the FIPA specifications.

In ActressMAS, the environment is responsible for passing the messages between agents. If remote messages are needed, they are routed through the containers and the server, and arrive at the destination environment. A message is also a wrapper similar to an ACL message, containing fields such as sender, receiver, the actual content and a conversation identifier. The content can be either a string or an object, which ensures a higher efficiency in some applications. The first word in a string message may be used as a performative, but this is not enforced in any way. In the example of the contract net protocol, a custom class is used for messages, which represents exactly the structure of an ACL message. The protocol is implemented according to the corresponding FIPA specification [32].

6.2. Analysis of ActressMAS in Comparison with Other Multi-Agent Frameworks

In this section we discuss how some features of other agent frameworks are related to the present characteristics of ActressMAS. This can help potential users to better assess the advantages and disadvantages of the proposed framework.

In terms of communication architecture, JADE [4] uses a peer-to-peer approach, while ActressMAS uses a client-server approach for the communication between containers. This is completely transparent from the agent's point of view, because an agent can simply decide to migrate to another container or send a remote message to an agent in another container, while the environment, the containers and the server carry out these actions.

From the point of view of agent scheduling, ActressMAS assigns all agents the same priority. The acting methods of all agents are executed in a turn. Different priorities could be imposed if some agents were ignored during some turns. So far, there has been no intention of introducing such mechanism. ActressMAS agents are lightweight, e.g., one can create one million agents on a computer without any special memory capabilities.

Since the BDI architecture is closely related to agent research, Jason [10] and Jadex [33] have integrated support for it. ActressMAS does not, but provides an example with the BDI architecture. Moreover, it does not contain an internal planning engine; the plans need to be created by the user.

Akka [18] allows a hierarchical organization of actors. An actor can be created by another actor which is then considered its "parent". Each parent can then supervise the execution of the tasks assigned to its children. ActressMAS agents are not organized in a hierarchy; all agents are implicitly on the same level. However, as implemented in the Skynet example, an agent can store a reference to another agent considered to be its parent and report to it, and conversely, parents can store references to their children. These references hold agent names, not object references.

Frameworks such as JaCaMo [34] and MaDKit [35] have integrated support for organizations. For example, in the Agent-Group-Role (AGR) organizational model, agents play roles in groups and create organizations. The roles define some constraints on the agent actions, i.e., obligations, interdictions, and permissions. Although an ActressMAS example models workflows defined as RADs, there is currently no explicit support for roles; however, this is envisioned for future work.

In ActressMAS, agents can create new agents and can indirectly destroy them, through the environment. Therefore, it is not designed for adversarial scenarios with agents belonging to different owners that try to directly harm opposing agents.

Although one of the main tenets of both actors and agents is loose coupling, ActressMAS allows a form of shared memory, especially useful when agents in a multi-agent simulation need to access a large environment. This *Memory* property of the environment should not be used to store global variables for the main logic of the agents, but when large amounts of data need to be accessed frequently, this solution is much more efficient than sending the data in the form of messages. A typical scenario that may benefit from this is when agents communicate indirectly through the environment, by stigmergy. The perceptual function of the agent may also be helped by the custom automatic filtering of agents using the observable properties of ActressMAS.

7. Conclusions

The paper gave an in-depth description of the architecture of ActressMAS, originally intended as a simple-to-use .NET multi-agent framework. However, it proved to be adequate for the implementation of various algorithms, protocols, and simulations, as shown by the example applications.

Considering the categories presented in Section 2, perhaps many types of software agent systems may be implemented using ActressMAS, such as simulations in the mobility, organizational, social, economic, or environmental domains. So far, ActressMAS has not been considered to be used for games or for multi-agent learning scenarios, e.g., supporting multi-agent reinforcement learning algorithms.

Throughout the development of the framework, simplicity has been a main goal. Several methods have been eliminated in successive versions, following the idea that "if it is not important to be in the product, it is important not to be in the product". It was also intended to help the user achieve what other frameworks or specifications provide, but without imposing any constraints. Especially, some standards may be useful for large systems where efficiency and interoperability are necessary, but they may not be needed for simpler, e.g., academic, protocols.

However, recommendations and examples are provided, which can guide the developer to create functionality supported by other frameworks, e.g., using FIPA ACL messages with performatives, implementing a Directory Facilitator, or designing agent-based applications with a BDI or reactive architecture.

In the future versions of ActressMAS, it should be established whether some of these features should be abstracted and integrated into the platform itself. But in this case, the developer should not be forced to use intrusive or mandatory features that he/she does not need.

For example, the concept of agent roles and explicit support for workflow modeling can be added. The platform can be extended with additional capabilities related to various agent architectures. For example, a mechanism to register rules with specified priorities can be incorporated for reactive applications. The planning part of the BDI architecture can be integrated by means of forward reasoning based on pattern matching, already included in the FunCs functional programming library [36] created by the author. Deductive reasoning can also be based on this library and employed within a logical agent architecture.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The ActressMAS framework and the implementations of the algorithms are open-source and fully available at: https://github.com/florinleon/ActressMas (accessed on 1 November 2021).

Conflicts of Interest: The author declares no conflict of interest.

Appendix A

This appendix provides additional information about the use of the proposed framework and the examples available in its GitHub repository [1].

As stated in the Introduction, ActressMAS was designed to simplify the teaching of multi-agent protocols and algorithms, and it has been used since 2018 in the author's department. In the following paragraphs, several applications are briefly presented. Even if some of them are inspired by different sources, all the implementations are original.

The example in Figure A1 uses a reactive architecture. It implements an idea from [37], where a swarm of robot vehicles (the blue circles) explores an unknown region searching for rock samples (the cyan squares). When a sample is found, it needs to be delivered to a central base (the red circle), which also provides a radio signal that can be used to estimate the direction and distance to it from any other location. Each vehicle has a hierarchy of simple behavioral rules, such as: 1. If an obstacle is detected, then change direction; 2. If carrying samples and being at the base, then drop samples; 3. If carrying samples and not being at the base, then travel up signal gradient; 4. If a sample is detected, then pick it up; 5. Otherwise, move randomly. The ordering also defines the priority of the rules: the rules with smaller numbers have a higher priority than those with higher numbers. Although these rules are very simple, the aggregated behavior of the swarm can be very complex, as vehicles can be seen searching and actively delivering samples to the central base. This is a typical example of emergent behavior in a multi-agent system.

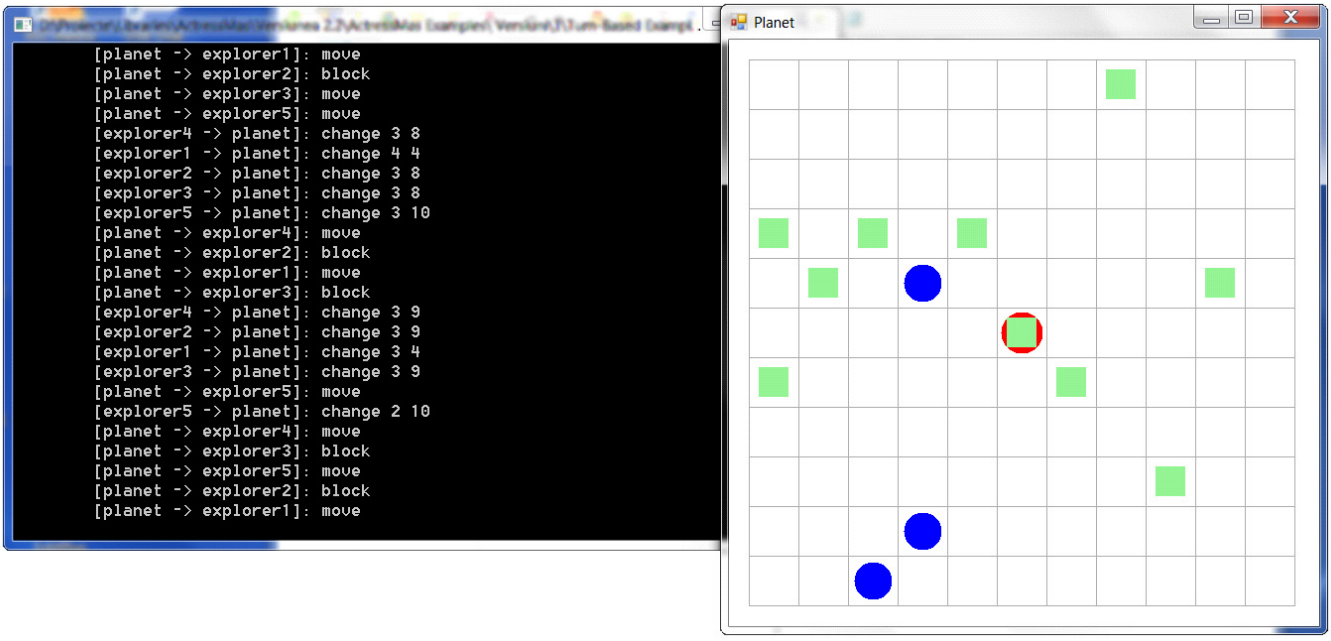

Figure A1. The implementation of a reactive architecture.

The next example implements a BDI architecture for agents [11] and is based on an idea from [38]. A helicopter, designated by a black ellipse in Figure A2, is patrolling for forest fires. Thus, its initial goals are to go from the first cell to the last cell and then reverse direction. When it detects a fire (the red rectangle) in its perceptual field, i.e., its current cell and two the adjacent ones, it adds a higher priority goal to get some water (the blue rectangle) from the first cell, move to the fire cell and drop the water. Therefore, the agent has beliefs such as: "position", "water", and "fire", and desires such as: "patrol right", "patrol left", and "extinguish fire". These goals are achieved by intentions, i.e., plans with a series of individual actions such as: "move left", "move right", "get water", and "drop water". The beliefs are updated based on the percepts received from the interaction with the terrain agent and its own actions.

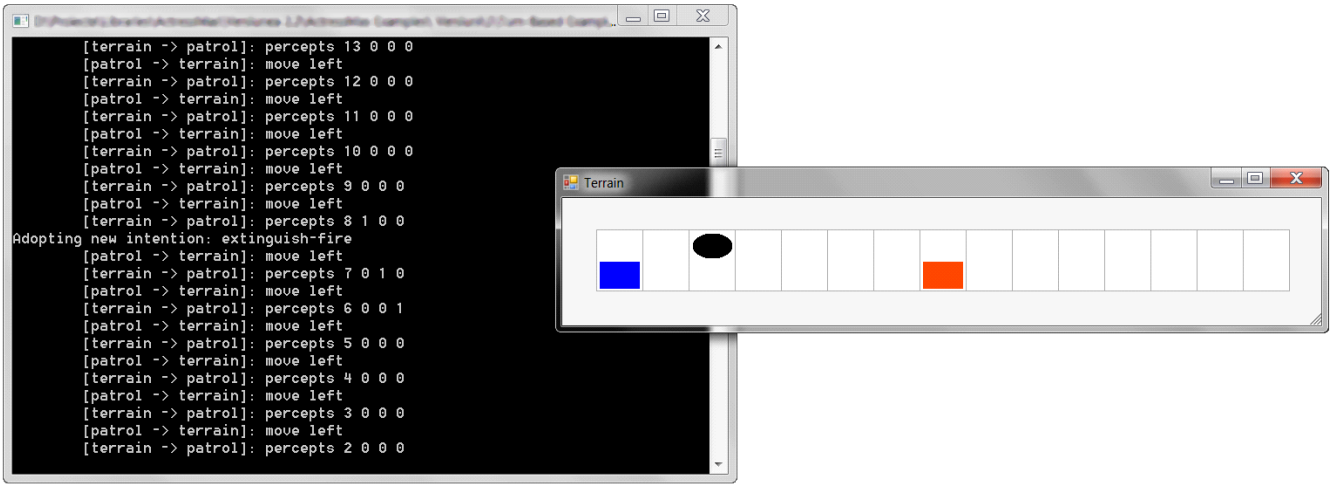

Figure A2. The implementation of a BDI architecture.

ActressMAS was also used to model business processes that define organizational activities. Such processes can be represented using role-activity diagrams [39]. The enactment of business agents for this purpose was previously achieved using the F# and Jason

programming languages [40]. The F# implementation, which used actors with "mailbox processors", was easily converted to ActressMAS.

The application whose GUI is displayed in Figure A3 is a simulation of the famous "game of life" [41]. The simulation is driven by three local rules for each cell: 1. Any living cell with two or three living neighbors survives into the next generation; 2. Any dead cell with three living neighbors becomes a living cell; 3. All other living cells die and all other dead cells remain dead.

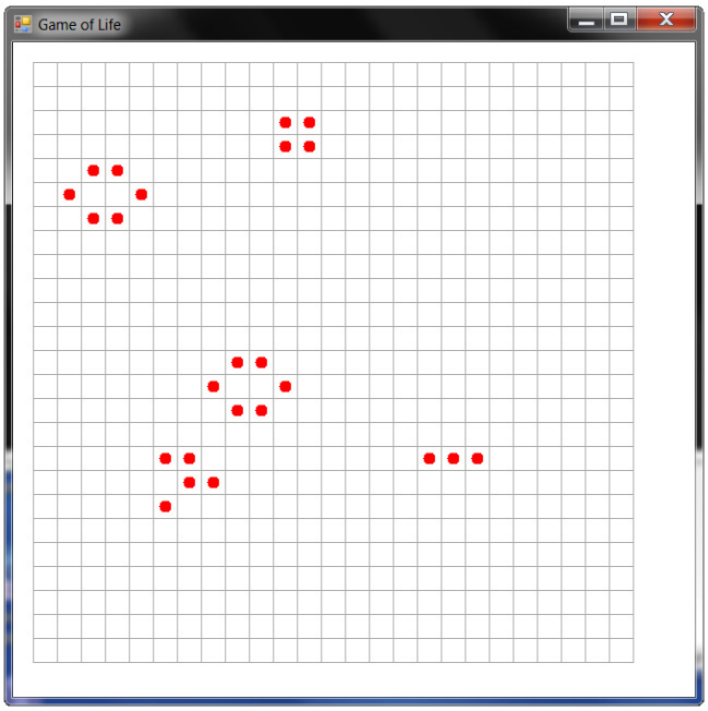

Figure A3. The implementation of the game of life using observables.

These rules can be easily implemented with observables, as explained in Section 4.2. Each cell is an agent that can register its position and its interest in perceiving only the neighboring cells in its Moore neighborhood, i.e., with eight surrounding cells. The state (living or dead) is also an observable property. Thus, each cell is aware of the state of its neighbors and can change its own state by applying the rules mentioned above.

ActressMAS was also used to simulate an environment with two species: predators (in this case, doodlebugs) and prey (in this case, ants). The rules of the simulation are as follows. An ant can: 1. Move randomly up, down, left, or right, if possible; 2. Breed: if an ant survives for three time steps, it creates an offspring in an adjacent cell, if it is free. Conversely, a doodlebug can: 1. Move: in each time step, the doodlebug moves to an adjacent cell containing an ant and eats it; otherwise, it moves randomly; 2. Breed: if a doodlebug survives for eight time steps, it creates an offspring; 3. Starve: if a doodlebug has not eaten an ant within three time steps, it dies.

For this simulation, the *Memory* property of the environment is used in order to store the ecological environment, with cells occupied by at most one type of insect. Again, each insect can perceive its neighboring cells, and when they need to reproduce, a new insect is created and added to the environment in an empty neighboring cell, if possible.

Figure A4 presents the result of the execution of a simulation, where one can see the oscillations of the two populations. This behavior has been theoretically modeled by the Lotka–Volterra equations.

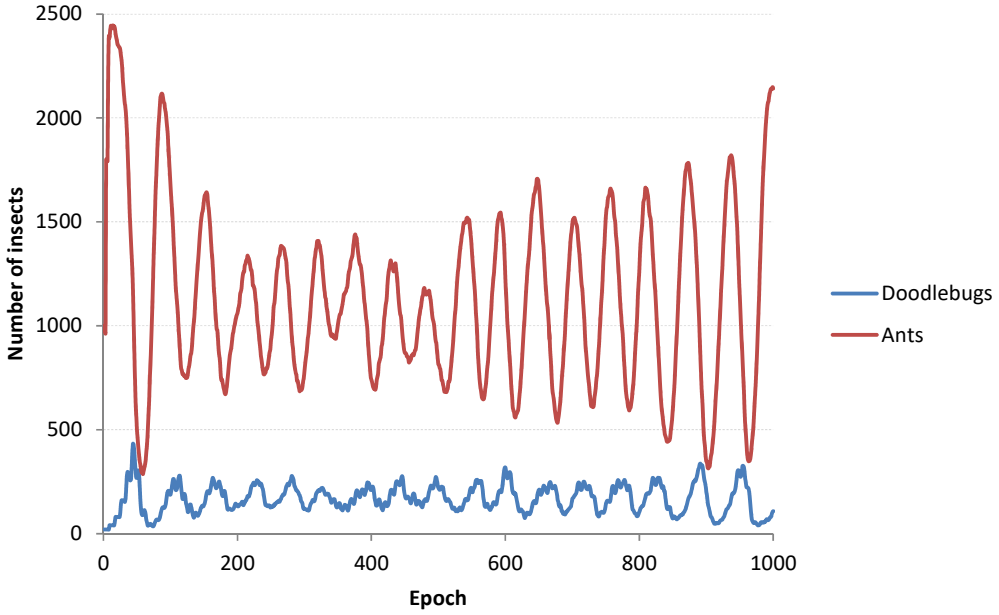

Figure A4. The results of a run of the predator-prey simulation.

Finally, a more complex simulation was made using the framework, i.e., a traffic simulator named "CarSim", which can be used to collect training data for a deep learning system intended for autonomous driving. The user can construct different types of road segments (a road segment is an agent) and place any number of cars (also agents) in different positions. As one can see in Figure A5, the white car with a black dot is the ego car (i.e., the autonomous vehicle), and the cars with other colors are the rest of traffic participants. The user can set several properties of the vehicles: the length, the initial speed, heading angle, acceleration, and the maximum speed, in order to simulate different driving behaviors (more cautious or more aggressive).

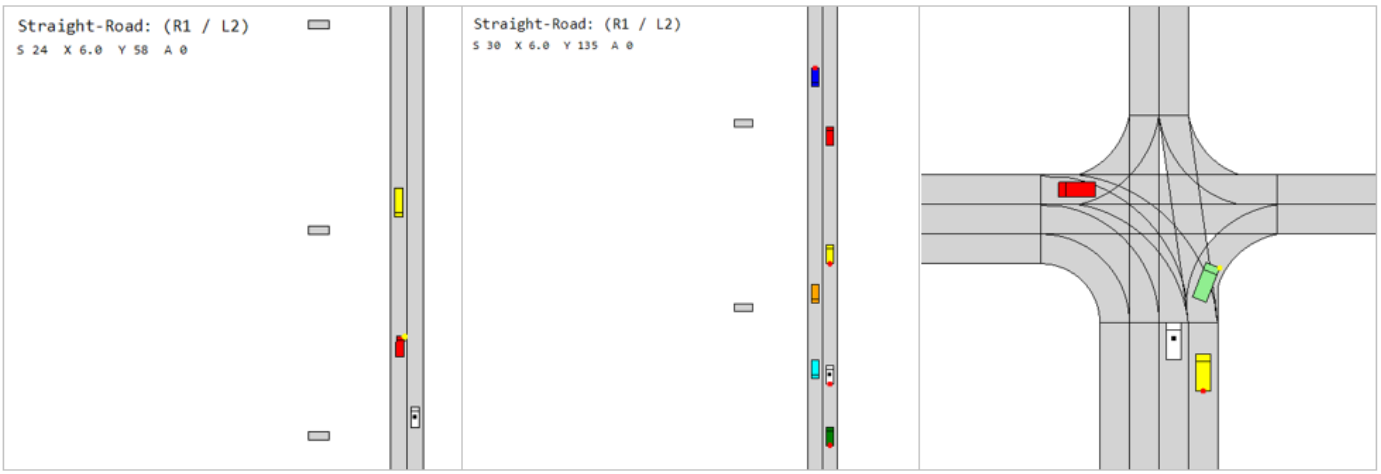

Figure A5. Different traffic simulations scenarios.

The simulator then computes the successive actions for each vehicle agent with a physics model combined with a symbolic model. An action is decided in each time step (e.g., each 0.2 s in the virtual time). The physics model is used to estimate the future trajectories of the traffic participants, based on their current positions, speeds and accelerations. The symbolic model contains rules to handle the interactions between agents. For example, if the physics model detects a possible collision, the vehicle slows down. If the maximum speeds of two or several vehicles require an overtaking to take place and if this is safe from

the physics point of view, the current agent begins the overtaking. After it is completed, the vehicle returns to its normal lane.

The simulator can also include static obstacles, bicyclists and pedestrians. Although the main perspective in the simulator is that of the ego car, each agent decides independently, but taking into account its surroundings, as stated above. From a simulation, data can be exported in order to be used in other learning contexts for trajectory prediction.

ActressMAS is provided in the form of a .NET dynamic-link library (DLL) which can be downloaded and to which a developer should add a reference in his/her project and use it directly. No other external packages are needed.

The GitHub repository [1] also offers a single C# solution with all the implemented examples. It can be opened and explored, e.g., using Visual Studio 2017 or newer. The version of the .NET framework is 4.7.2. Table A1 presents the full list of available examples with the main concepts they address.

Table A1. A summary of the examples available in the GitHub repository of ActressMAS v3.0 (as of December 2021).

Project Name	Purpose and Learning Points
Simple Examples→Agents1, Agents2, Agents3, Agents4, Agents5	These examples show how agents should be created in ActressMAS and how to set up the environment. They show different possible execution types: parallel or sequential. The main focus is on sending messages and processing them in the *Act* method. The use of *ActDefault* method at the end of a turn with no received messages is also presented.
Simple Examples→ MultipleMessages	It shows a system where messages are exchanged between several worker agents and a manager agent in a way that would make it difficult for some messages to be delivered if the message passing infrastructure were not properly designed. This is also a test case for ActressMAS.
Simple Examples→ SendingObjects	It shows how user-defined objects can be directly sent in the content of messages.
Reactive Architecture	It presents an implementation of the reactive architecture where multiple behaviors can be activated based on the current state of the agent. The behaviors have priority levels, such that only the behavior with the highest priority defines the next action.
BDI Architecture	It presents an implementation of the Belief-Desire-Intention (BDI) architecture, where agents have explicit state information (beliefs), can have goals (desires) and make plans to achieve these goals (intentions).
LRTA Search	It presents an implementation of the Learning Real-Time A* (LRTA*) path finding algorithm. The search is designed as a continuous conversation between the search agent and the map agent, where the search agent is informed in each state about the neighboring states and the value of the heuristic function in that state. It reflects the behavior of an agent that discovers the map dynamically while performing the search.
Shapley Value	It presents a multi-agent system with worker agents with different skill values solving tasks with different difficulty levels. They divide their payoffs according to their marginal contributions to solving the tasks, using the game theoretic solution concept of the Shapley value.
Auctions→English with broadcast, English without broadcast	These projects implement the English auction protocol in two variants: when all the bids are broadcast to all the agents and when the bidders communicate only with the auctioneer, which in turn communicates the current best price after each round of bids.
Auctions→Vickrey	It implements the Vickrey auction, a sealed-bid protocol where the highest bidder is the winner but pays the second price. The dominant strategy of this auction protocol is bidding the true valuation, thus no agent has a motivation to bid a higher or a lower amount.

Table A1. *Cont.*

Project Name	Purpose and Learning Points
Yellow Pages	It presents a multi-agent system with service providers, clients (service consumers) and a service broker. Service providers can register or deregister at any time. This is the functionality envisioned by the FIPA Directory Facilitator.
Zeuthen Strategy	It implements the Zeuthen bargaining strategy based on the risks of breaking down the negotiation. The agent that has more to lose if the negotiation fails should be more willing to concede. At each step, the agent with a smaller risk needs to make a concession big enough to change the balance of risks, such that the other agent should concede in the next round.
Contract Net Protocol	It implements the FIPA specification for the contract net protocol. The agents communicate by messages that conform to the FIPA ACL structure. This example features some virtual postmen that should deliver letters and may exchange some of them in order to optimize their routes. Thus, it includes a heuristic travelling salesman problem solver as a subcomponent. The contract net protocol is used to find the (near-)optimal task allocation in terms of payoffs associated with letter delivery and the costs associated with tour length.
Mechanism Design	It implements the Clarke–Groves tax system which eliminates the incentive of an agent to lie about its true preference in a majority voting scenario.
Iterated Prisoner Dilemma	It implements the iterated prisoner dilemma game, which is a simple model that reveals deep questions related to human selfishness and cooperation. It includes multiple response strategies whose outcomes can be compared: acting randomly, always defecting, and "tit-for-tat" (where an agent first cooperates, then chooses the action chosen by the other agent in the previous round).
Predator-Prey→ PredatorPreyConsole, PredatorPreyGui	These projects present a simulation with two species, predators and prey, which emphasizes the oscillating evolution of the two populations. The grid world (i.e., the natural environment) is stored in the *Memory* property of the ActressMAS software environment and can be accessed by the individuals. Individuals can be created and removed from the environment.
Simple Observables→ ColorGame, NumberGame	These projects focus on the use of observable properties. Each agent perceives the other agents with a certain assigned color or with an assigned number in a specific range.
Game of Life	It is an implementation of Conway's "game of life" using observable properties of agents/cells to compute the number of alive neighbors.
Voting	It presents a voting protocol that first tries to identify the Condorcet winner using the Copeland's method, and if no Condorcet winner exists, it uses Borda count to produce the result.
Mobile Examples→ MyServerConsole, MasMobileConsole, MyServerGui, MasMobileGui	These projects present an example with mobile and static agents. A mobile agent visits the existing containers, collects information form the static agents running there and then returns to its original container and reports the information. There are two equivalent implementations, one using a console for displaying messages, i.e., a character user interface, and another using a Windows-based graphical user interface.
Mobile Examples→ RemoteMessages	It shows how two agents in different containers can communicate by sending remote messages.
Benchmarks→PingPong	This is a benchmark that measures the number of messages that can be exchanged between agents in a time unit in a communication-intensive scenario.
Benchmarks→Skynet, SkynetNumeric	This is a benchmark that measures the performance of agent creation and basic calculations. In Skynet, the numbers are encoded into string messages, while in SkynetNumeric, the numbers are encoded as 64-bit integer numbers. However, ActressMAS does not exhibit a significant difference in performance in the two cases.

References

1. Leon, F. ActressMAS Library. 2018–2021. Available online: https://github.com/florinleon/ActressMas (accessed on 1 November 2021).
2. Leon, F. ActressMAS. A .NET Multiagent Framework. 2021. Available online: http://florinleon.byethost24.com/actressmas (accessed on 1 November 2021).
3. Pal, C.V.; Leon, F.; Paprzycki, M.; Ganzha, M. A Review of Platforms for the Development of Agent Systems. *arXiv* **2020**, arXiv:2007.08961; 40 pages.
4. Bellifemine, F.L.; Caire, G.; Greenwood, D. *Developing Multi-Agent Systems with JADE*; Wiley Series in Agent Technology: Chichester, UK, 2007.
5. Luke, S.; Balan, G.C.; Panait, L.; Cioffi-Revilla, C.; Paus, S. MASON: A Java Multi-Agent Simulation Library. In Proceedings of the Agent 2003 Conference on Challenges in Social Simulation, Chicago, IL, USA, 2–4 October 2003.
6. Bernstein, P.; Bykov, S.; Geller, A.; Kliot, G.; Thelin, J. *Orleans: Distributed Virtual Actors for Programmability and Scalability*; Technical Report MSR-TR-2014-41; Microsoft: Redmont, WA, USA, 2014.
7. Ritter, F.E.; Tehranchi, F.; Oury, J.D. ACT-R: A cognitive architecture for modeling cognition. Wiley Interdisciplinary Reviews. *Cogn. Sci.* **2019**, *10*, e1488. [CrossRef]
8. Laird, J.E. *The Soar Cognitive Architecture*; MIT Press: Cambridge, MA, USA, 2012.
9. Newell, A.; Shaw, J.C.; Simon, H.A. Elements of a theory of human problem solving. *Psychol. Rev.* **1958**, *65*, 151–166. [CrossRef]
10. Bordini, R.H.; Hübner, J.F.; Wooldridge, M. *Programming Multi-Agent Systems in AgentSpeak Using Jason*; Wiley Series in Agent Technology: Chichester, UK, 2007.
11. Rao, A.S.; Georgeff, M.P. Modeling Rational Agents within a BDI-Architecture. In Proceedings of the 2nd International Conference on Principles of Knowledge, Representation and Reasoning, Cambridge, MA, USA, 22–25 April 1991; pp. 473–484.
12. Rao, A.S. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96), Einhoven, The Netherlands, 22–25 January 1996.
13. Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; Zaremba, W. OpenAI Gym. *arXiv* **2016**, arXiv:1606.01540; 4 pages.
14. Beattie, C.; Leibo, J.Z.; Teplyashin, D.; Ward, T.; Wainwright, M.; Küttler, H.; Lefrancq, A.; Green, S.; Valdés, V.; Sadik, A.; et al. DeepMind Lab. *arXiv* **2016**, arXiv:1612.03801; 11 pages.
15. Wilensky, U.; Rand, W. *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NETLogo*; The MIT Press: Cambridge, MA, USA, 2015.
16. Dosovitskiy, A.; Ros, G.; Codevilla, F.; Lopez, A.; Koltun, V. CARLA: An Open Urban Driving Simulator. *arXiv* **2017**, arXiv:1711.03938; 16 pages.
17. Horni, A.; Nagel, K.; Axhausen, K.W. (Eds.) *The Multi-Agent Transport Simulation MATSim*; Ubiquity Press: London, UK, 2016. [CrossRef]
18. Bonér, J.; Klang, V.; Kuhn, R. Akka Library. 2021. Available online: https://akka.io (accessed on 1 November 2021).
19. Petabridge. Akka.NET Library. 2021. Available online: https://getakka.net (accessed on 1 November 2021).
20. Hewitt, C.; Bishop, P.; Steiger, R. A Universal Modular Actor Formalism for Artificial Intelligence. In Proceedings of the 3rd International Joint Conference on Artificial intelligence (IJCAI'73), Stanford, CA, USA, 20–23 August 1973; pp. 235–245.
21. Wooldridge, M. *Intelligent Agents. Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence*; Weiss, G., Ed.; The MIT Press: Cambridge, MA, USA, 2000; pp. 27–77.
22. Wooldridge, M. *An Introduction to Multiagent Systems*, 2nd ed.; Wiley: Hoboken, NJ, USA, 2009.
23. Burgin, M. Systems, Actors and Agents: Operation in a Multicomponent Environment. *arXiv* **2017**, arXiv:1711.08319v1; 28 pages.
24. Rumbaugh, J.; Jacobson, I.; Booch, G. *Unified Modeling Language Reference Manual*, 2nd ed.; Pearson Education: Boston, MA, USA, 2005.
25. Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification. 2002. Available online: http://www.fipa.org/specs/fipa00037/SC00037J.html (accessed on 1 November 2021).
26. Austin, J.L. *How to Do Things with Words*; Clarendon Press: Oxford, UK, 1975.
27. Cugola, G.; Ghezzi, C.; Picco, G.P.; Vigna, G. Analyzing Mobile Code Languages. In *Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 1997; Volume 1222, pp. 94–109.
28. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Professional: Boston, MA, USA, 1994.
29. Asynkron, A.B. Proto. Actor Library. 2021. Available online: https://proto.actor (accessed on 1 November 2021).
30. Foundation for Intelligent Physical Agents. FIPA Abstract Architecture Specification. 2002. Available online: http://www.fipa.org/specs/fipa00001/SC00001L.html (accessed on 1 November 2021).
31. Poslad, S.; Buckle, P.; Hadingham, R. The FIPA-OS agent platform: Open source for open standards. In Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents, Manchester, UK, 10–12 April 2000; Volume 355.
32. Foundation for Intelligent Physical Agents. FIPA Contract Net Interaction Protocol Specification. 2002. Available online: www.fipa.org/specs/fipa00029/SC00029H.html (accessed on 1 November 2021).
33. Braubach, L.; Pokahr, A.; Lamersdorf, W. Jadex: A BDI-Agent System Combining Middleware and Reasoning. In *Whitestein Series in Software Agent Technologies*; Birkhäuser-Verlag: Basel, Switzerland, 2006; pp. 143–168. [CrossRef]

34. Boissier, O.; Bordini, R.H.; Hübner, J.F.; Ricci, A.; Santi, A. Multi-agent Oriented Programming with JaCaMo. *Sci. Comput. Program.* **2013**, *78*, 747–761. [CrossRef]
35. Gutknecht, O.; Ferber, J. The MadKit Agent Platform Architecture. In *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*; Springer: Berlin/Heidelberg, Germany, 2001; pp. 48–55. [CrossRef]
36. Leon, F. FunCs Library. 2018–2021. Available online: https://github.com/florinleon/FunCs (accessed on 1 November 2021).
37. Steels, L. Cooperation Between Distributed Agents Through Self-Organisation. In Proceedings of the IEEE International Workshop on Intelligent Robots and Systems, Towards a New Frontier of Applications, Ibaraki, Japan, 3–6 July 1990; pp. 8–14. [CrossRef]
38. Taillandier, P.; Gaudou, B.; Grignard, A.; Huynh, Q.; Marilleau, N.; Caillou, P.; Philippon, D.; Drogoul, A. Building, composing and experimenting complex spatial models with the GAMA platform. *Geoinformatica* **2019**, *23*, 299–322. [CrossRef]
39. Ould, M.A. *Business Process Management: A Rigorous Approach*; British Computer Society, Meghan Kiffer: Tampa, FL, USA, 2005.
40. Leon, F.; Bădică, C. A Comparison Between Jason and F# Programming Languages for the Enactment of Business Agents. In Proceedings of the International Symposium on INnovations in Intelligent SysTems and Applications (INISTA), Sinaia, Romania, 2–5 August 2016. [CrossRef]
41. Martin, G. The Fantastic Combinations of John Conway's New Solitaire Game "Life". *Math. Games. Sci. Am.* **1970**, *223*, 120–123. [CrossRef]