Chapter 16

Benny Lutati, Inna Gontmakher, Michael Lando, Arnon Netzer, Amnon Meisels, and Alon Grubshtein

Abstract Applications of multi-agent system (MAS) are versatile. In this chapter we focus on a specific application domain—agent-oriented programming for distributed constraint reasoning (DCR). The field of DCR deals with constraint-based problems that are distributed among multiple agents. The agents need to arrive at an optimal solution to the global combinatorial problem, and in order to do so, they run a distributed search algorithm. Another important aspect of MAS software development is MAS simulation. In this regard, this chapter introduces a new agent-based research tool for designing and testing DCR algorithms. The new tool—AgentZero—is specifically designed for the specification, implementation, and evaluation of DCR search algorithms. AgentZero provides full support to researchers of distributed constraints algorithms in the form of an extensive agent-based environment for algorithmic research that includes a distributed run-time environment, built-in performance measures that are automatically used by all algorithms, and visualization tools that help design and understand the behavior of complex distributed search algorithms. The API of the AgentZero simulator is described in detail and important architectural decisions that enable analysis and smooth implementation of a variety of algorithms are explained and described. In the context of AOSE, this chapter exemplifies two aspects: agent-based simulation environment and tools, and a variety of development and runtime aids for agent-based systems.

Keywords Multi-agent system simulator • Distributed algorithms • Multi-agent visualization • Multi-agent algorithm development • Multi-agent algorithm evaluation environment

B. Lutati • I. Gontmakher • M. Lando • A. Netzer • A. Meisels • A. Grubshtein
Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel
e-mail: benny.lutati@gmail.com; netzerar@cs.bgu.ac.il; am@cs.bgu.ac.il; alon.grubshtein@gmail.com

O. Shehory and A. Sturm (eds.), Agent-Oriented Software Engineering, DOI 10.1007/978-3-642-54432-3_16, © Springer-Verlag Berlin Heidelberg 2014
1 Introduction

A multi-agent system (MAS) is a distributed system composed of multiple autonomous agents. An agent is an independent software entity, which strives to reach a goal; the goal can be either private or shared by several agents. To reach its goal, an agent can communicate with other agents by using message passing. An agent has its own local memory, and in most cases the information that an agent holds can be classified into the following categories: public data—which the agent is willing to share with other agents; private data—which the agent would like to keep for itself; and tradable data—which is kept private but can be traded with other agents with the purpose of influencing them to perform a desired action.

Agent-oriented systems have many applications; in the robotics field, for example, the agent-oriented approach is used to coordinate robots performing different actions without interfering with one another. In the navigation field—applications like Waze [1] are a MAS designed to help drivers navigate by sharing information available to different agents. Managing and working with big data is also a field where the agent-oriented approach plays a crucial role. The Map Reduce paradigm where different agents process different parts of the data in order to extract needed information is becoming the de facto framework for storing and processing massive data. Another good example is the distributed search field where several agents attempt to optimize a solution to a common problem, such as meeting, scheduling, and resource allocation.

As an example of the latter, consider multiple departments at a university, each constructing its weekly schedule for the semester. The schedules of many departments are constrained because the curricula of groups of students include courses from these “constrained departments.” The agents constructing their departmental schedules need to arrive at an optimal solution to the global schedule problem, and in order to do so, they run a distributed search algorithm. The family of such distributed search algorithms for solving distributed constraints problems is termed distributed constraint reasoning (DCR) algorithms and is the focus of the present chapter.

MAS applications deployment and testing are inherently complicated. Distributed programming is a resource-consuming task in terms of time and cost. For this reason, simulation tools form a very important measure to be used when developing MAS applications. This chapter presents an innovative multi-agent simulation environment that is tailored specifically for designing, implementing, and testing DCR search algorithms.

The outline of the rest of this chapter is as follows. The next section covers several common settings and properties of different MASs, as well as the AgentZero MAS simulator and research tool. Afterwards, the DCR framework is shortly introduced and an application of distributed search is explained. An implementation of a DCR algorithm using AgentZero will follow together with a short explanation of the simulation behind the scenes. The user front-end is then briefly reviewed and the chapter concludes with a comparison of AgentZero to different tools that are available in the MAS domain.
2 Common MAS Settings and Properties

A common MAS architecture (Fig. 16.1) entails implementing every agent as a process on a different computer. Another alternative suggests implementing a group of agent processes executing on the same machine or several groups on multiple machines. In such cases, the agents communicate with one another via message passing. Indeed, architectures for MASs commonly include a message passing mechanism in support of agent communication.

2.1 Agents Cooperation and Privacy

A typical distributed multi-agent setting assumes that the knowledge of agents about their environment is limited in scope. Agents can influence the actions of one another using message passing by exposing information, performing negotiation, etc. The cooperation level of the agents varies. On one end there are cooperative agents that can be generally described as agents that agree to follow a common protocol for the sake of a shared goal, while on another end, self-interested agents can be thought of as agents that try to maximize their own goal solely.

2.2 Execution Modes

An important attribute of any MAS is its mode of synchronization. There are two primary modes of execution: asynchronous, and synchronous. The asynchronous mode is the more general case. In this mode, there is no assumption of synchronization between the agents. The sent messages will get to their destination eventually.
As in TCP, there are no assumptions on the message ordering between different agents, but in case that an agent sends two messages one after the other to the same recipient—the message that was sent first will arrive first. This mode is similar to the way the Internet network operates. The synchronous mode is an iterative-like mode; in this mode the agents operate in a round-by-round fashion. Any message sent at round $i$ will reach its destination at round $i+1$. Agents are aware of the round at which they operate and are assumed to process and respond to all messages from round $i$ before round $i+1$ starts. In a real network, a synchronous execution can be achieved using different synchronization techniques. This type of execution is applicable in many cases, including computer games and online bidding.

3 MAS Simulation via AgentZero

AgentZero is a java-based easy to use, extendible MAS simulator and research tool. It provides a simulated agent execution environment and application analysis support for distributed problem solving. The framework specializes in DCR algorithms design and evaluation but can be used to simulate many other MAS applications too. AgentZero provides an intuitive programming API. It has the capabilities to simulate a multi-agent environment on a single computer and to attach many types of statistical profilers and debug assisting tools into the simulated environment. Another important aspect incorporated into the framework is the AgentZero web laboratory—a collaboration area for researchers, which is designed to provide a simple location for sharing experimental results, algorithms, and modules in order to enhance the ease and productivity of research efforts. In this respect, AgentZero is a good example of practical MAS development and engineering aids.

3.1 AgentZero Modularity

In order for a simulator to be useful for a wide range of applications, it must be capable of supplying generic services, which can be applied to any agent-oriented application implemented on top of it. AgentZero’s design goal was to define the widest possible range of generic services that can be automatically used by all implemented applications and especially DCR algorithms, which will be covered in the next section. These services, applied to a wide range of applications with different inherent structures, are the basis for the usability of the simulator as a research tool for investigating and performing comparative evaluation of agent-oriented driven applications. The principles according to which AgentZero was designed are listed below:

- Simple agent design—following simple API, the agent has access to a set of comprehensive tools and can be implemented in a way that is indifferent to the
execution mode. Agents have the ability to expose properties that are later to be set in the experiment designing process, fundamental actions of the agents are automatically monitored by the system in order to provide useful statistics—this reduces to minimum the need to write code that is unrelated to the actual agent behavior.

- Powerful message passing mechanism—messages are passed by value and not by reference and are conveniently handled in an RPC fashion; statistical message delays can be applied in order to measure the application’s behavior under different network conditions without slowing down the actual execution.
- Both asynchronous and synchronous executions are supported out of the box.
- Generic performance measurements—Measuring performance of a distributed application is a crucial and challenging task. For example, the total execution time of a distributed algorithm in a simulated environment provides little information when the number of agents exceeds the number of concurrent computational cores in the system. Providing a measure of all nonconcurrent actions is both technically challenging and prone to mistakes. Providing standard, application-independent measurements, the implementation architecture must be designed so that aggregating this information is hidden from the application implementation and can easily be extended for gathering additional information.
- Incomplete information—A typical distributed multi-agent setting assumes that the knowledge of agents about their environment is limited in scope. In simulated agent-based environments (such as AgentZero), this limitation is enforced by the framework, leading to realistic implementation of algorithms and applications.
- General purpose tools and modules—As more algorithms and applications are introduced, the number of reoccurring parts used as building blocks increases. These can include a set of preprocessing procedures, common data structures, and benchmark problem structures. Providing such services as tools and pluggable modules do more than just simplifying the work of application and algorithm designers; it also generates a common ground for the reference research community to facilitate comparison among implementations.

4 Distributed Constraint Reasoning

DCR is a specific field of research of MASs [2, 3]. Consider, for example, a large hospital that is composed of many wards. Each ward constructs a weekly timetable assigning its nurses to shifts. The construction of a weekly timetable involves solving a constraint satisfaction or optimization problem for each ward. Some of the nurses in every ward are qualified to work in the Emergency Room. Hospital regulations require a certain number of qualified nurses (e.g., for Emergency Room) in each shift. This imposes constraints among the timetables of different wards. Assigning an unqualified nurse to some shift is acceptable only if there are enough qualified nurses assigned to that shift in the other wards. A natural model for this multi-agent combinatorial problem is a distributed constraint satisfaction problem.
(DCSP) [14, 17] or distributed constraint optimization problem (DCOP) [15, 16], in which the agents represent the different wards, the value assignments of agents are schedules and the requirement to have at least one qualified nurse per shift among all wards is an example of a constraint between agents.

Formally, DCR problems are composed of a set of agents. Agents contain variables with finite domains of values and are connected by constraints among their variables. A solution to a DCOP is a global assignment for variables that minimizes the costs of all constraints in the system. DCR algorithms are usually distributed search algorithms in which all agents cooperate in the search for a globally consistent or optimal solution. The solving procedure involves assignments of all agents to all their variables and exchange of information among all agents, to check the consistency of assignments with constraints among agents [4–6]. The DCR model assumes that the data held by agents cannot be centralized. Most studies of DCR search algorithms consider the motivation for this assumption to be privacy: consider the hospital example again, in which head nurses of different wards are assigning nurses to shifts while considering the personal constraints of the nurses in each ward. Head nurses would not want to reveal the personal data of nurses in their ward and their own personal preferences and considerations. The only information that a head nurse needs to reveal regarding her ward is the time slots in which she is assigning an Emergency Room qualified nurse, while keeping all other information private.

4.1 Example of a DCR Algorithm

Let us now step through an example of a (very) simple DCR algorithm and see how it proceeds. The simple algorithm is called: synchronous back tracking (SBT) [7]. The algorithm assumes that constraints map joint assignments to either “satisfiable” or “inconsistent” constraints, which makes the problem a DCSP. The SBT approach for solving this type of problem is the simplest possible, but for simplicity, we also introduce several common assumptions:

- Every agent handles only one variable (which means that in this simple example an agent and a variable can be treated as the same thing).
- The domain of the variables is represented as a set of natural numbers.
- Only binary constraints are available in the given problem.
- The agents are lexicographically ordered. That is, agent i precedes agent j in the total ordering if $i < j$.

Given the previous assumptions, the following is the SBT pseudocode:

The algorithm’s starting point can be seen in the initialize method. This method is the first to be called by each agent; any other operations that the agent may perform will be in response to receiving a message. The first agent initializes a variable called CPA, which carries a consistent tuple of the assignments of the agents it passed so far. The first agent initializes the search by creating a CPA, assigning its variable on
the CPA and sending the CPA to the next agent. Every agent that receives the CPA
tries to assign its variable without violating constraints with the assignments on the
CPA. If the agent succeeds to find such an assignment to its variable, it appends
the assignment to the tuple on the CPA and sends it to the next agent. If it cannot
find a consistent assignment, it sends the CPA back to the previous agent to change
its assignment, thus performing a chronological backtrack. An agent that receives
a backtrack message removes the assignment of its variable and tries to reassign it
with a consistent value. The algorithm ends successfully if the last agent manages
to find a consistent assignment for its variable. The algorithm ends unsuccessfully
if the first agent encounters an empty domain. [8]

5 Agent Implementation Example

Accessibility and ease of use is a fundamental requirement of any application
framework. Algorithm designers often require a simple and clean API, which
abstract away information on the execution environment. This enables the designer
to focus on the algorithms’ logic and not on the environment.

The AgentZero programming API is designed to match these clear requirements.
Written in Java, Fig. 16.3 presents an SBT implementation corresponding to the
pseudocode shown in Fig. 16.2. The previously mentioned pseudocode assumptions
still apply and most of the algorithm’s pseudocode can be directly translated into
the programming API.

6 Agent Implementation

The following covers the part of the API related to agent implementation as can be
seen in Fig. 16.3. Agents in AgentZero extend the SimpleAgent abstract class—this
class provides an extensive domain-specific language for agents and the following
features:

– **Single initialization point**—every agent defines a start method, which will get
  automatically called as soon as the agent is started.
– **Message sending**—using the API: send(<message-name>, <message-
  args>),to*(...). This send method is a variadic method, which returns a
  mediation object that is initialized with the message name and a package of the
  message arguments. The mediation object has many useful “to*” methods like
toNextAgent, toNeighbors, etc. By calling these methods, the actual message is
  passed to the mailer.
– **Message to method binding**—in the above example, one may notice the usage
  of the @WhenReceived(<message-name>) annotation. This syntax will cause
  the agent to automatically call the annotated method when a message with name
equals to the name specified in the annotation arrives. The message arguments are unpackaged into the annotated method while invoking it—this behavior resembles RPC behavior, which makes it very easy to manage messages.

- **Termination signaling**—instead of handling the termination of the agent itself, the SimpleAgent includes many useful “finish*” methods, which automatically broadcast a termination signal. Due to the fact that AgentZero is a simulated MAS environment, several of the “finish*” methods also support submitting the algorithm’s results, which can then be tested for correctness or aggregated into a final report.
public class SBTAgent extends SimpleAgent {
    Assignment cpa;

    public void start() {
        if (this.isFirstAgent()) {
            cpa = new Assignment();
            assignCPA();
        }
    }

    void assignCPA() {
        for (Integer v : getDomain()) {
            if (cpa.isConsistentWith(getId(), v, getProblem())) {
                cpa.assign(getId(), v);
                if (cpa.isFull()) finish(cpa);
                else send("CPA", cpa).toNextAgent();
                return;
            }
        }
    }

    backtrack();
}

void backtrack() {
    if (isFirstAgent()) finishWithNoSolution();
    else send("BACKTRACK").toPreviousAgent();
}

@WhenReceived("CPA")
void handleCPA(Assignment cpa) {
    this.cpa = cpa;
    assignCPA();
}

@WhenReceived("BACKTRACK")
void handleBacktrack() {
    cpa.unassign(getId());
    backtrack();
}

Fig. 16.3 SBT implementation with AgentZero programming API
6.1 Technical Aspects of Simulated Execution

AgentZero simulates a complex MAS network using threads. In the following section, a technical overview of the synchronous execution simulation is described. To build a local agent memory, all messages that are passed between threads are automatically deep-copied. The first step of an AgentZero execution is to configure the environment requested according to a specific configuration file; this enables and configures modules in the environment. This first step entails initializing all the requested modules and generating databases according to the statistic collectors (if this was not already done by previous executions). The next AgentZero step is creating agent runners; there are worker threads that are each responsible for the execution of one or more agents. In synchronous execution, the number of worker threads is the same as the number of CPU cores that the executing computer has. This setup was tested to produce the most efficient and fast execution. After the agent runner threads are up, a structure called agent-states is initialized. This is a shared structure between all the agent runners. This data structure coordinates the agent runners during the execution. Figure 16.4 shows a schematic diagram of the synchronous execution.

As shown in Fig. 16.4, at each round (a.k.a. iteration), the agent runners iterate on the agent states data-structure. By performing the atomic CPU instruction—CAS (Compare and Swap)—on the atomic Boolean, which guards each agent
state, the agent runner insures that it is the only thread that currently manages the specific agent state. While managing the agent state, the agent runner will drive the agent to handle all of the requests that are in its current round queue. The agent’s queue is a two-layer queue. One belongs to the current round, and the other to the next round. When the agent takes the next message, it will retrieve it from the current round queue, and when the agent sends a message, the mailer will put it in the next round queue of the recipient agent. When the agent is done handling all of its current round messages, the agent runner will continue to iterate and try to drive the next unhandled agent. At the end of the agent states data-structure, there is a barrier. Every agent runner will eventually reach the barrier and when the last runner will enter the barrier, the system clock will be triggered. The system clock will notify its listeners of the end of the round. One of these listeners is the mailer, which has a pointer to all of the agent queues. At the end of the round, the mailer will flip each agent’s current round queue with the next round queue and release the barrier, after which this whole process will start over again and continue to do so until the algorithm is done.

### 6.2 Execution Environment Modules

In AgentZero, agent implementation is decoupled from the execution environment it is running in; thus, every algorithm can be executed on each of these modes. In every environment, one can associate a set of modules, which can generate special problem types, test the solution or the way to the solution, collect wanted statistics, visualize the algorithm progress and many more module types. Although AgentZero comes bundled with many module implementations, the module implementations themselves are not closely coupled with the system and can be contributed via third-party implementers. Every module has its own simple API for creating your own variant of the module. In fact, many of the modules that AgentZero comes bundled with were contributed by the BGU DCR group members while working with the tool. AgentZero has an extensive documentation where you can find all the standard module names. From the AgentZero lab website,¹ one can download and contribute more modules. The modules, similar to the agent definition, are decoupled from the execution environment. The combination of all these modules and algorithms assembles an experiment. This makes the creation of the modules and algorithms very easy, as one needs not care about the execution environment’s internal architecture and its comprising components.

¹The complete AgentZero software package is available to DCR researchers and students at our DCR homepage [http://www.cs.bgu.ac.il/~dcr](http://www.cs.bgu.ac.il/~dcr). It is routinely used by all members of the DCR research group at BGU. In addition, it is the main tool by which graduate students studying distributed constraints algorithms implement their final projects.
7 Simulation Front End

An important aspect that is often missed or neglected by other research tools and MAS frameworks is the front end and the ease of use both for new and advanced users. AgentZero walks the extra mile by providing simple and powerful front end for developers and researchers.

7.1 Simulation Results Analysis

During the development phase of a problem-solving MAS, the developer has certain needs that cannot be fully satisfied without a proper user interface. For example, to optimize the algorithm, the developer has a need to watch statistical data immediately after each execution. To debug the algorithm, it can be helpful to be able to see the problem, view run-time logs, and even sometimes look at visualizations to better understand the way in which the algorithm works. AgentZero attempts at addressing all these needs via an interactive Eclipse plugin. Using the plugin, it is easy to set up an AgentZero project, creating agents and modules, and viewing the execution data. The plugin also gives the opportunity to use a better debugging mode by automatically saving failed scenarios and providing the possibility to reexecute them with a debugger attached.

7.2 Simulated Environment Visualization

One of the best ways for understanding how an algorithm (or an application) works is to observe visualizations that represent different points of view on the algorithm’s execution. An important feature of AgentZero is its ability to visualize the simulated environment. Visualizing an application execution is a two-step process—analysis step and visualization step. The first step includes monitoring an execution by watching the side-effects it induces on the environment and marking important events. The second step focuses on organizing these events on a timeline and presenting these events to the user. As a large simulated environment that is executed on a single core machine will incorrectly translate simulation time into real world time, special measures are taken to produce visualizations that are close to the way the application will operate on a real distributed MAS. Figure 16.5 shows one of the available visualizations that comes with AgentZero—the network traffic visualization that allows the user to understand the impact each agent has on the simulated network.
8 Comparison to Related MAS Simulators

AgentZero is a new addition to the DCR research field. Two different DCR specialized MAS simulators already exist, DisChoco and FRODO. In the following, we briefly describe these tools and then compare them to AgentZero and MASS, which is a general MAS simulation tool.

8.1 DisChoco

DisChoco [9] is a Java platform for solving DCR problems developed by Redouane Ezzahir and Mohamed Wahbi—it came with relatively extensive library of problem generators and uses an internal communication simulator. It also has the ability to deploy the agents in a real distributed system.

8.2 FRODO

FRODO [10] is a Java open-source framework for distributed combinatorial optimization, initially developed at the Artificial Intelligence Laboratory (LIA) of École Polytechnique Fédérale de Lausanne (EPFL), Switzerland. FRODO comes with several built-in algorithms and a suite of problem generators for benchmarking. It also support out-of-the-box small number of statistics relevant to DCR.
8.3 MASS

The multi-agent simulation suite (MASS) [15] is a software package intended to enable modelers to utilize the tools of agent-based simulation in various fields, without having to develop heavy programming skills. In the context of this chapter, MASS is a generic MAS simulation that provides a new programming language and IDE that is used to define the behavior of the agent. It than allows relatively easy implementation of an environment for the agent to execute on top and contains several tools to configure this environment and to examine execution results and statistics.

In Table 16.1, we compare the different multi-agent simulation tools along various functionality, usability, and practical criteria. Our observations indicate that AgentZero, although generic and applicable to many domains, specifically introduces benefits for the DCR simulation domain. Note that AgentZero supports most of the features provided by existing simulation tools and also provides a simple, extensive, and better streamlined user experience and research facilities.

9 Summary

This chapter refers to multi-agents systems that support DCR and its search algorithms. In particular, it presents a new simulation tool—AgentZero—for distributed search algorithms for DCOPs. The complexity of DCR algorithmic simulation and its consequent importance is briefly discussed and an example of an agent implementing distributed optimization search was presented. The DCR field is an important example for agent-oriented programming that includes a complex algorithmic component.

In order to alleviate the burden of design and testing of DCR algorithms, a simulation tool like AgentZero is a necessity. In its core, AgentZero is an agent-based simulator that specializes in simulating a DCR-oriented environment. It exposes a relatively simple and intuitive API for implementing agents and has the ability to measure the efficiency of the behavior of agents following the algorithmic protocol in the simulated environment. AgentZero provides a solution for researchers investigating the behavior of different algorithms as it enables comparative simulated studies without the need to set up a large costly and complicated network of agents. For this reason, it serves as one of the main research tools at the BGU DCR group and it is used by students in several DCR-related courses. One can say that MAS simulators are an essential tool for every MAS development and testing environment and not only specific to the DCR area. One may also comment on the need for them to be adjustable and extensible, as suggested by AgentZero. A lot has been learned while developing and applying AgentZero. On the technical side, we have learned about different models for simulation of MAS
<table>
<thead>
<tr>
<th>Functionality</th>
<th>DisChoco</th>
<th>FRODO</th>
<th>MASS</th>
<th>AgentZero</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic statistic collection</td>
<td>Yes, but limited</td>
<td>No</td>
<td>No</td>
<td>Yes, by using hooks</td>
</tr>
<tr>
<td>Execution modes—supported out of the box</td>
<td>Asynchronous</td>
<td>Asynchronous and Synchronous</td>
<td>Synchronous</td>
<td>Asynchronous and Synchronous</td>
</tr>
<tr>
<td>Extendibility of the system (without changing the actual framework code)</td>
<td>No</td>
<td>Possible, but complicated</td>
<td>Very extendible via programming and in some cases via wizards</td>
<td>Very extendible, many module types available</td>
</tr>
<tr>
<td>Out-of-the-box statistics, execution visualizations and tools</td>
<td>Single primitive visualization, several statistics</td>
<td>No visualizations, several statistics</td>
<td>Contains a suite for defining statistics and visualizations</td>
<td>Several visualizations, many statistics. More statistics can be added as modules, currently there is no API for adding more visualizations</td>
</tr>
<tr>
<td>Possibility to write non-DCR-related agents</td>
<td>Based on Choco, which is a general MAS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Simulation of local memory on local execution</td>
<td>No, sharing pointers to messages</td>
<td>No, sharing pointers to messages</td>
<td>No</td>
<td>Achieved by deep-copying the messages automatically</td>
</tr>
<tr>
<td>Support for execution timeouts</td>
<td>Unknown</td>
<td>Yes—only in local mode</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Tools for analyzing results</td>
<td>The UI shows some options but they are not implemented</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

(continued)
<table>
<thead>
<tr>
<th>Feature</th>
<th>DisChoco</th>
<th>FRODO</th>
<th>MASS</th>
<th>AgentZero</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability</strong></td>
<td>Agent Programming Language</td>
<td>Java and XML</td>
<td>FABLES (dedicated language)</td>
<td>Java</td>
</tr>
<tr>
<td><strong>Documentation</strong></td>
<td>No</td>
<td>Yes, including online tutorial</td>
<td>Complete, also offers courses for paying customers</td>
<td>Yes, including online tutorial</td>
</tr>
<tr>
<td><strong>Learning curve</strong></td>
<td>Without documentation—very steep</td>
<td>Steep</td>
<td>Moderate-steep, requires learning a new programming language and for some operations Java knowledge is still required</td>
<td>Moderate</td>
</tr>
<tr>
<td><strong>Open source</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>The source is available for paying customers</td>
<td>Yes</td>
</tr>
<tr>
<td>Requires modifying the source in order to add algorithms or applications</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>User interface installation and overall usage</strong></td>
<td>Desktop application—mostly non-functional, frequently crashes</td>
<td>Desktop application—many important features missing, not developer friendly</td>
<td>Eclipse integration for development and desktop application for working with the execution itself. Overall, the UI looks complete</td>
<td>Eclipse plug-in worked well. The desktop user interface is helpful for new users</td>
</tr>
<tr>
<td><strong>Domain specific</strong></td>
<td>Ability to reproduce an experiment</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ability to run large experiments</td>
<td>Yes, supports execution locally, on local cluster and grid middleware</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Agent per thread only, thus limited size problems</td>
<td>Yes, supports execution locally, on local cluster and grid middleware</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automatic problem and scenario generation—built-in and external</td>
<td>Built-in problem and scenario generation—built-in and external</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Built-in support for receiving problems in a dedicated format</td>
<td>Only built-in or by DisChoco</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Does coding an agent with the framework resemble the pseudocode</td>
<td>Partially, with FABLES, the code highly resembles the pseudocode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ease of creating an experiment</td>
<td>Very easy, simple XML structure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Portable problem format</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Support for simulation of message delays, message corruption, or message loss</td>
<td>FABLES does not have fixed communication scheme, one would have to implement those features by himself as part of the model that can be rather difficult</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
environments. The design and development process has started from the simple but resource wasteful—agent per thread model and ended up by developing our own “reaction model” where threads can help one another in driving several agents together.

From the software engineering viewpoint, the goal was to develop a framework that is modular enough so that it will be able to simulate any environment and agents. Several ways to modularize the execution environment were investigated (e.g., OSGi—Alliance, OSGi. OSGi service platform, release 3. IOS Press, Inc., 2003.) But found too complex for our target audience, the final decision was to design a new module system, such that will be easily understandable for our users. From our users, we learned that the most difficult task related to writing applications on top of a MAS (a.k.a. agents) is the debugging process. Having a locally simulated environment makes it possible to use general-purpose debugging tools. Still, this task remains difficult.

AgentZero was already used in research setting [12–13]. AgentZero is continuously maintained, and we intend to keep extending it. One example is the addition of support for a dynamic execution environment, where agents and agent properties can be dynamically changed over time. Another example includes the addition of built-in support for scenario logging and reconstruction—a feature that we hope will reduce the complexity of debugging the agents’ execution.

References