Agents Assembly: Domain Specific Language for Agent Simulations

Przemysław Hołda[1][0000−0001−5425−7440], Kajetan
Rachwał[1][0000−0003−1524−7877], Jan Sawicki[1][0000−0002−8930−7564], Maria
Ganzha[1][0000−0001−7714−4844], and Marcin Paprzycki[2][0000−0002−8069−2152]

[1] Faculty of Mathematics and Information Science, Warsaw University of Technology,
Warsaw, Poland {przemyslaw.holda,kajetan.rachwal}.stud@pw.edu.pl
[2] Systems Research Institute Polish Academy of Sciences, Warsaw, Poland

Abstract. Researchers studying group behavior, social dynamics, or epidemiology lack an easy-to-use tool to run large-scale simulations. This contribution introduces a domain specific language (Agents Assembly; AASM) and a toolset for creating and running scalable simulations, using containerized environment. The proposed language supports describing abstract concepts, such as agent, message, and behavior. Its structure, resembling assembly mnemonics, is simple to understand. The language is supported by a code generation module with a graphical user interface, which allows defining simulations using instruction blocks. The AASM code is translated to Python and runs in a distributed containerized ecosystem, utilizing a slightly modified SPADE agent framework. The developed toolset includes also data storage and live visualization.

Keywords: Domain specific language · Multi-agent simulations · Scalable computing.

1 Introduction

Multi-agent simulations have been used in multiple fields, ranging from supporting electoral campaigns [3] to analyzing the spread of viruses in a population [1]. However, creating such simulations requires both domain knowledge and programming skills (to code the simulation). It may hinder the potential use of simulations, as two different skills are required. Moreover, the development of large-scale agent-based simulations requires knowledge of the agent-based paradigm and tools, which are not common skills. For all practical purposes, only the Repast (for High Performance Computing [3]) is a mature tool capable of running large-scale agent simulations. However, earlier experiences with Repast [2] showed that its use is not simple. To address this problem, a domain specific language (DSL) called Agents Assembly (AASM) was developed. It allows users without advanced programming skills to define the structure and behavior of a multi-agent system. The proposed language is complemented with a Runtime Environment, which facilitates running and analyzing simulations formulated using

---

[3] https://repast.github.io

AASM. Moreover, since AAMS is executed on top of containerized instances of the Smart Python Agent Development Environment (SPADE) [4] agent platform, large-scale simulations can be realized.

2   Main purpose

The starting point of this contribution was the authors' own experience in using a general-purpose programming language (Python) and an agent framework (SPADE) to create agent-based simulations. Upon reflection, it became apparent that the process may be difficult for researchers without an extensive computer science background. Hence, the idea of developing an environment that delivers "the same capabilities" as SPADE but without the need for in-depth knowledge of programming came to be. As a result, more researchers from all fields should be able to use multi-agent-based simulations. In this context, the following objectives were identified by analyzing the process of developing large-scale agent-based simulations.

1. To allow users to define simulations using simple logic, arithmetic, and concepts of the simulation domain (e.g., agent, behavior).
2. To create a user-friendly environment in which a simulation may be prepared, run, and analyzed.
3. To ensure that large-scale simulations (consisting of thousands of agents) can be executed in an architecture that is simple to set up.

To satisfy these objectives, the proposed ecosystem consists of: (1) domain specific language (AASM), (2) runtime that translates it to Python and runs on a (slightly modified) SPADE platform, (3) possibility to run multiple SPADE instances, in separate dockerized containers, on networked computers, (4) user-friendly front-end, (5) data storage (for result persistence and further analysis), and (6) live visualization.

3   Demonstration

The authors will now describe the Agents Assembly DSL and its ecosystem. The main features of the proposed approach will be presented using two simulations during the live demonstration.

3.1   Agents Assembly – key concepts

Agents Assembly has been developed based on analysis of literature and own experiences gathered when developing agent-based simulations. The language structure is divided into agents, messages, and a network, which creates a skeleton, upon which a multi-agent simulation will be realized.

---

[4] `https://github.com/javipalanca/spade`

The **agent** instruction defines agents. The AASM allows declaring the structure and the initial state of agents using a carefully selected set of parameters. Parameters may be numbers, enums, lists of connections, numbers, or messages. Enums and numbers may be initialized using set values or random values drawn from the specified distribution. Inside the agent scope behaviors are defined with the **behav** keyword. An agent's behavior is a sequence of actions performed by the agent when certain conditions are met. The language supports running behaviors during agent setup, after a specified delay, periodically, and upon receiving a message from another agent.

The definition of an action starts with the **action** keyword. Two types of actions are supported: modifying the internal state and sending messages to other agents. The body of an action includes a list of instructions to be executed. AASM includes more than thirty instructions, such as mathematical operations, control statements, and array procedures. Their style was heavily inspired by the design of the assembly mnemonics.

Listing 1.1: Autonomous car AASM listing

```
agent autonomous_car
    prm current_speed, float, dist, uniform, 0, 120
    behav speed_communication, cyclic, 15
        action inform_cars, send_msg, speed_alert, inform
            set send.speed, current_speed
            send connections
        eaction
    ebehav
eagent
```

The code in Listing 1.1 defines the blueprint for the agent `autonomous_car`, which has one parameter and one behavior. Parameter `current_speed` is a floating-point number, initially drawn from the uniform distribution. The agent's behavior (`speed_communication`) runs cyclically. Inside it, there is an action `inform_cars`, used to send messages of a specific type (`speed_alert inform` defined in Listing 1.2). The first instruction sets the message parameter `speed` with the value of `current_speed`. Finally, the message is sent to the list `connections` that represent the recipient agents. The initial content of the list `connections` depends on the agent network definition (see Listing 1.3).

AASM supports defining messages, using the **message** instruction. They are identified by their name and an ACL performative (as specified in the FIPA00037J Standard) [5]. Messages contain numerical parameters.

Listing 1.2: Speed alert

```
message speed_alert, inform
    prm speed, float
emessage
```

---
[5] http://www.fipa.org/specs/fipa00037/SC00037J.html

Code snippet in Listing 1.2 declares a message `speed_alert` with the performative `inform` and a parameter `speed`.

The **graph** keyword is used to describe the graph structure of the agent network. In its scope, the connections of each agent are defined.

Listing 1.3: Agent network

```
graph statistical
    defg autonomous_car, 300, dist_exp, 0.1
egraph
```

The code in Listing 1.3 defines a network of type `statistical`. This name refers to the fact that the size of each agent's initial connection list is drawn from a statistical distribution while the connections are assigned randomly. Here, for agents of type `autonomous_car`, the numbers are drawn from the exponential distribution, with $\lambda = 0.1$. The network's size totals 300, and `autonomous_car` instances constitute the complete simulation environment.

To facilitate the use of the AASM, a translator capable of validating an AASM program, and generating a run-time code, was developed. Specifically, a target-agnostic intermediate representation is created first. Next, it is translated to Python code, for the SPADE agent framework. Interestingly, the Python code generated from the snippets above is approximately six times longer than the AASM code. Note that the translation, using the intermediate representation, delivering code for other agent platform(s) can be instantiated.

The language structure is, by design, simple and block-like. It allows for easy generation of code with graphical user interface (GUI) modules. A custom AASM code generator can be seen in Figure 1.
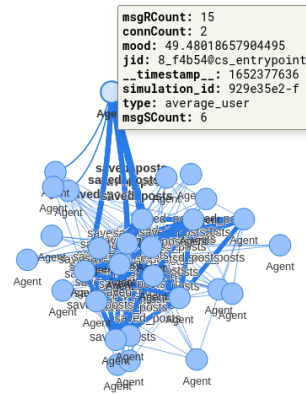

Fig. 1: GUI code generator

Fig. 2: Simulation visualization

3.2   Implementing AASM ecosystem

The AASM ecosystem consists of three main components: the Interface, the Simulation Run Environment (SRE), and the Communication Server. Each component corresponds to a Docker [6] Swarm stack, running multiple microservices. Through the Interface, a user may define a simulation using the GUI code generator, run it and analyze it using management and visualization modules. The Interface interacts with SRE mainly through HTTP requests (although Bolt Protocol [7] connection with the database is possible). When a user creates an AASM simulation, SRE translates it, generates the network structure, and distributes agents between Agent Containers. Agent Containers are dockerized microservice processes that run the translated code using the SPADE agent platform. Each container holds the number of simulation agents assigned by the Simulation Load Balancer (one of the microservices in the SRE stack). Agents send their states to the database through Kafka [8], which ensures that the database does not get overloaded by write requests. Agents send messages through the Communication Server, an instance (or a cluster, as they may be freely scaled) of Tigase XMPP Servers [9]. The containerized approach allows an easy increase in the computational resources used by the simulation.

During the experiments, the system was deployed on a cluster of 15 physical computers, running 120 Agent Containers. It was also deployed on 5 virtual machines, running 5 Agent Containers each. A single agent container is capable of running multiple agents. Experimental results indicated a limit of around 1000 agents per container and a good (stable) performance at 100 agents (tested on Intel Core i7-6700HQ @ 2.6GHz and 24GB DDR4 RAM). The exact number of agents depends on agents' complexity (number of operations per minute) and the hardware.

In the system, the Interface may be accessed from any node in the dockerized network. Its connection to a graph database (neo4j [10]) grants live access to the simulation data (values and aggregates of agent and message parameters). The agents' state is visualized directly through a dedicated module that presents an interactive graph of data available in the database (depicted in Figure 2). The visualization module also supports live plotting of data streams during the simulation. Data can be requested directly through Cypher (neo4j's query language) queries or via GUI query creator providing a user-friendly alternative.

3.3   Demonstrations

During the demonstration, the attendees will be able to see running simulations prepared in AASM. They will also have the possibility to modify the details of the simulations through the code generating GUI, observing its simplicity.

---

[6] https://docker.com

[7] https://boltprotocol.org

[8] https://kafka.apache.org

[9] https://tigase.net

[10] https://neo4j.com

The presentation will be performed on 2 computers, starting with 2 agent containers and scaling to 8. The simulations will be verified in terms of the validity of the results and the system's stability.

4   Concluding remarks

The purpose of the work was to ease the development of agent-based simulations. In this context, a domain specific language, Agents Assembly, was proposed. The design of Agents Assembly was based on the evaluation of actual user needs. Thanks to translation to an intermediate code, it is possible to develop support for other agent platforms and frameworks.

The system is being tried by a group of students at the Warsaw University of Technology to develop a complex traffic simulation. Their initial feedback has been positive. However, it was suggested that the number of available mathematical functions should be increased, which resulted in adding built-in **sin**, **cos**, **log**, and **pow** instructions. Their experiences, collected at the end of the semester, will be used to further improve the AASM ecosystem.

Separately, during experiments, it has been noticed that the system's stability can be affected in the case of vast numbers of connections. Specifically, during a simulation (running on 15 networked workstations) with an agent network with more than 399 million connections (large, dense, all-to-all graph), congestion in the system has been identified. However, it should be stressed that even when the communication bottleneck has been detected, the system did not crash. Nevertheless, this problem requires addressing in future development.

The current version of the Agents Assembly ecosystem can be found at `https://agents-assembly.com`. The web page gives access to an online version of the translator, links all pertinent repositories, and the language documentation. Contributions to its further development are welcome and requested.

References

1. Castro, B.M., de Abreu de Melo, Y., Fernanda dos Santos, N., Luiz da Costa Barcellos, A., Choren, R., Salles, R.M.: Multi-agent simulation model for the evaluation of COVID-19 transmission. Computers in Biology and Medicine **136** (2021)
2. Ciecierski, J., Mai, V.B., Słupczyński, M., Zyskowski, W.: Multi-agent simulation of the world found in the G. R. R. Martin's novel "Sandkings". In: Ganzha, M., Maciaszek, L., Paprzycki, M. (eds.) Position Papers of the 2015 Federated Conference on Computer Science and Information Systems. Annals of Computer Science and Information Systems, vol. 6, pp. 249–256. PTI (2015). https://doi.org/10.15439/2015F417, `http://dx.doi.org/10.15439/2015F417`
3. de Sola Pool, I., Abelson, R., Popkin, S.: Candidates, Issues and Strategies: A Computer Simulation of the 1960 and 1964 Presidential Elections. M.I.T. paperback series, Massachusetts Institute of Technology Press (1965)