Rafael H. Bordini, Louise A. Dennis and Yves Lespérance (eds)
Pre-proceedings of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS 2019)
13th-14th May 2019, Montreal, Canada
cgi.csc.liv.ac.uk/~lad/emas2019/accepted/EMAS2019_paper_21.pdf

JS-son - A Minimal JavaScript BDI Agent Library

Timotheus Kampik and
Juan Carlos Nieves

Umeå University, 901 87, Umeå, Sweden
{tkampik,jcnieves}@cs.umu.se

Abstract. There is a multitude of agent-oriented software engineering frameworks available, most of them produced by the academic multi-agent systems community. However, these frameworks often impose programming paradigms on their users that are hard to learn for engineers who are used to modern high-level programming languages such as JavaScript and Python. To show how the adoption of agent-oriented programming by the software engineering mainstream can be facilitated, we provide an early, simplistic JavaScript library prototype for implementing belief-desire-intention (BDI) agents. The library focuses on the core BDI concepts and refrains from imposing further restrictions on the programming approach. To illustrate its usefulness, we demonstrate how the library can be used for multi-agent systems simulations on the web, as well as embedded in Python-based data science tools.

Keywords: Reactive Agents · Belief-Desire-Intention · Multi-agent Systems.

1 Motivation

A multitude of multi-agent system (MAS) frameworks has been developed [2] by the scientific community. However, these frameworks are rarely applied outside of academia, likely because they require the adoption of design paradigms that are fundamentally different from industry practices and do not integrate well with modern software engineering toolchains. A recent expert report on the status quo and future of *engineering multi-agent systems*[1] concludes that "[m]any frameworks that are frequently used by the MAS community–for example Jason and JaCaMo–have not widely been adopted in practice and are dependent on technologies that are losing traction in the industry" [4]. Another comprehensive assessment of the current state of agent-oriented software engineering and its implications on future research directions is provided in Logan's *Agent Programming Manifesto* [3]. Both the *Manifesto* and the EMAS report recommend developing agent programming languages that are easier to use (as one of several ways to facilitate the impact of multi-agent systems research). The EMAS report highlights, *in particular*, the need to provide libraries that better integrate with technologies and programming paradigms that are relevant in the industry and points to the success this approach has in the machine learning community.

In this demonstration, we take a first step towards addressing the aforementioned recommendations. We follow a pragmatic and minimalistic approach: instead of creating a comprehensive multi-agent systems framework, we create *JS-son*, a light-weight

[1] The report was assembled as a result of the EMAS 2018 workshop.

library that can be applied in the context of existing industry technology stacks and toolchains and requires little additional, MAS-specific knowledge.

Design Approach

Programming languages like Lisp and Haskell are rarely used in practice but have influenced the adoption of (functional) features in mainstream languages like JavaScript and C#. It is not uncommon that an intermediate adoption step is enabled by external libraries. For example, before JavaScript's `array.prototype.includes` functor was adopted as part of the ECMA Script standard[2], a similar functor (`contains` and its aliases `include` / `includes`) could already be imported with the external library `underscore`[3]. Analogously, JS-son takes the belief-desire-intention (BDI) [5] architecture as popularized in the MAS community by frameworks like Jason [1] (as the name *JS-son* reflects) and provides an abstraction of the BDI architecture as a *plug and play* dependency for a widely adopted programming language.

2 Library Description

The library provides object types for creating agent and environment objects, as well as functions for generating agent beliefs, desires, intentions, and plans [4].
The **agent** realizes the BDI concepts as follows:

Beliefs: A belief can be any JSON object or JSON data type (string, number, array, boolean, or null).
Desires: Desires are generated dynamically by agent-specific desire functions that have a desire identifier assigned to them and determine the value of the desire based on the agent's current beliefs.
Intentions: A `preference` function filters desires and returns *intentions* - an array of JSON objects.
Plans: The plan's *head* specifies which intention needs to be active for the plan to be pursued. The body specifies how the plan should update the agent's beliefs and determines the actions the agent should issue to the environment.

Also, each agent has a `next()` function to run the following process:

1. It applies the belief update as provided by the environment (see below).
2. It applies the agent's preference function that dynamically updates the intentions based on the new beliefs; i.e., the agent is *open-minded*.
3. It runs the plans that are active according to the updated intentions, while also updating the agent beliefs (if specified in the plans).

[2] `https://www.ecma-international.org/ecma-262/7.0/#sec-array.prototype.includes`
[3] `https://underscorejs.org/#contains`
[4] The library–including detailed documentation, examples, and tests–is available at `https://github.com/TimKam/JS-son`.

4. It issues action requests that result from the plans to the environment.

Alternatively, it is possible to implement simpler belief-plan agents; i.e., as a plan's head, one can define a function that determines–based on the agent's current beliefs–if a plan should be executed.

The **environment** contains the agents, as well as a definition of its own state. It executes the following instructions in a loop:

1. It runs each agent's `next()` function.
2. Once the agent's action request has been received, the environment processes the request. To determine which update requests should, in fact, be applied to the environment state, the environment runs the request through a filter function.
3. When an agent's actions are processed, the environment updates its own state and the beliefs of all agents accordingly. Another filter function determines how a specific agent should "perceive" the environment's state.

3 Potential Use Cases

We suggest that JS-son can be applied in the following use cases:

Data science. With the increasing relevance of large-scale and semi-automated statistical analysis ("data science") in industry and academia, a new set of technologies has emerged that focuses on pragmatic and flexible usage and treats traditional programming paradigms as second-class citizens. JS-son integrates well with Python- and Jupyter notebook[5]-based data science tools, as shown in Demonstration 1.

Web development. Web front ends implement functionality of growing complexity; often, large parts of the application are implemented by (browser-based) clients. As shown in Demonstration 2, JS-son allows embedding BDI agents in single-page web applications, using the tools and paradigms of web development.

Education. Programming courses are increasingly relevant for educating students who lack a computer science background. Such courses are typically taught in high-level languages that enable students to write working code without knowing all underlying concepts. In this context, JS-son can be used as a tool for teaching MAS programming.

Internet-of-Things (IoT) Frameworks like Node.js[6] enable the rapid development of IoT applications, as a large ecosystem of libraries leaves the application developer largely in the role of a system integrator. JS-son is available as a Node.js package.

4 Demonstrations

We provide two demonstrations that show how JS-son can be applied.

[5] `https://jupyter.org/`.
[6] https://nodejs.org/

JS-son meets Jupyter. The first demonstration shows how JS-son can be integrated with data science tools, i.e., Python libraries and Jupyter notebooks[7]. As a simple proof-of-concept example, we simulate opinion spread in an agent society and run an interactive data visualization.

JS-son on the Web The second demonstration presents a JS-son port of *Conway's Game of Life*. It illustrates how JS-son can be used as part of a web frontend[8]. The demonstration makes use of JS-son's simplified belief-plan approach.

5 Limitations and Future Work

This demonstration paper presents an *early stage* library prototype that provides minimalistic abstractions for implementing BDI agents. We concede that the prototype provides–so far–merely minimally viable abstractions to instantiate agents. To increase the library's relevance for researchers, teachers, and practitioners alike, we propose the following work:

Extend library interface. It makes sense to enable JS-son agents and environments to act in distributed systems and communicate with agents of other types, without requiring extensive customization by the library user. A possible way to achieve this is supporting the open standard agent communication language FIPA ACL[9]. However, as highlighted in a previous publication, FIPA ACL does not support communication approaches that have emerged as best practices for real-time distributed systems like *publish-subscribe*. To facilitate JS-son's reasoning abilities, interfaces to machine learning tools and best practices for developing learning JS-son agents can be developed.

Move towards real-world usage. In this demonstration paper, we provide two proof-of-concept examples of JS-son agents. To demonstrate the feasibility of JS-son, it is necessary to apply the library in advanced scenarios. Considering the relatively small technical overhead JS-son agents imply, the entry hurdle for a development team to adopt JS-son is low, which might facilitate early real-world adoption.

Implement a Python port. While JS-son can be integrated with the Python ecosystems, for example via Jupyter notebooks, doing so implies technical overhead and requires knowledge of two programming languages. To facilitate the use of BDI agents in a data science and machine learning context, we propose the implementation of *Py_son*, a Python port of JS-son.

Acknowledgements This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

[7] The Jupyter notebook is available on GitHub at `http://s.cs.umu.se/lmfd69` and on a Jupyter notebook service platform at `http://s.cs.umu.se/girizr`.

[8] The simulation is available at `http://s.cs.umu.se/chfbk2`.

[9] `http://www.fipa.org/specs/fipa00061/index.html`

References

1. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology). John Wiley & Sons, Inc., USA (2007)
2. Kravari, K., Bassiliades, N.: A survey of agent platforms. Journal of Artificial Societies and Social Simulation **18**(1), 11 (2015)
3. Logan, B.: An agent programming manifesto. International Journal of Agent-Oriented Software Engineering **6**(2), 187–210 (2018)
4. Mascardi, V., Weyns, D., Ricci, A., Earle, C.B., Casals, A., Challenger, M., Chopra, A., Ciortea, A., Dennis, L.A., Díaz, Á.F., Fallah-Seghrouchni, A.E., Ferrando, A., Fredlund, L.Å., Giunchiglia, E., Guessoum, Z., Günay, A., Hindriks, K., Iglesias, C.A., Logan, B., Kampik, T., Kardas, G., Koeman, V.J., Larsen, J.B., Mayer, S., Méndez, T., Méndez, T., Nieves, J.C., Seidita, V., Tezel, B.T., Varga, L.Z., Winikoff, M.: Engineering Multi-Agent Systems: State of Affairs and the Road Ahead. SIGSOFT Engineering Notes (SEN) (January 2019)
5. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In: Allen, J., Fikes, R., Sandewall, E. (eds.) Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning. pp. 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA (1991)