JS-son - A Lean, Extensible JavaScript Agent Programming Library

Timotheus Kampik(✉) and Juan Carlos Nieves

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

Abstract. A multitude of agent-oriented software engineering frameworks exist, most of which are developed by the academic multi-agent systems community. However, these frameworks often impose programming paradigms on their users that are challenging 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 a lean JavaScript library prototype for implementing reasoning-loop agents. The library focuses on core agent programming concepts and refrains from imposing further restrictions on the programming approach. To illustrate its usefulness, we show how the library can be applied to multi-agent systems simulations on the web, deployed to cloud-hosted function-as-a-service environments, and embedded in Python-based data science tools.

Keywords: Reasoning-loop agents · Agent programming · Multi-agent systems

1 Introduction

Many multi-agent system (MAS) platforms have been developed by the scientific community [11]. However, these platforms 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 tool chains. A recent expert report on the status quo and future of *engineering multi-agent systems*[1] concludes that "many 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" [13]. 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* [12]. 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 following issues:

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

L. A. Dennis et al. (Eds.): EMAS 2019, LNAI 12058, pp. 215–234, 2020.
https://doi.org/10.1007/978-3-030-51417-4_11

1. The tooling of academic agent programming lacks maturity for industry adoption. In particular, Logan states that "there is little incentive for developers to switch to current agent programming languages, as the behaviours that can be easily programmed are sufficiently simple to be implementable in mainstream languages with only a small overhead in coding time" [12].
2. Recent trends towards higher-level programming languages have found little consideration by the multi-agent systems community. In contrast, the machine learning community has embraced these programming languages, for example by providing frameworks like Tensorflow.js for JavaScript [16] and Keras for Python [6].
3. Consequently, agent programming lacks strong industry success stories.

Based on these challenges, the following research directions can be derived:

1. Provide agent programming tools that offer useful abstractions in the context of modern technology ecosystems/software stacks, without imposing unnecessarily complex design abstractions or niche languages onto developers.
2. Embrace emerging technology ecosystems that are increasingly adopted by the industry, like Python for data science/machine learning and JavaScript for the web.
3. Evaluate agent programming tools in the context of industry software engineering.

While this work cannot immediately provide practical agent programming success stories, it attempts to provide a contribution to the development of tools and frameworks that are conceptually pragmatic in that they limit the design concepts and technological peculiarities they impose on their users and allow for a better integration into modern software engineering ecosystems. We follow a pragmatic and lean approach: instead of creating a comprehensive multi-agent systems framework, we create *JS-son*, a light-weight library that can be applied in the context of existing industry technology stacks and tool chains and requires little additional, MAS-specific knowledge.

The rest of this chapter is organized as follows. The design approach for JS-son is described in Sect. 2. The architecture of JS-son, as well as the supported reasoning loops, are explained in Sect. 3. Subsequently, Sect. 4 explains how to program JS-son agents using a small, step-by-step example. Section 5 elaborates on scenarios, in which using JS-son can be potentially beneficial; for some of the use case types, simple proof-of-concept examples are presented in Sect. 6. Then, JS-son is put into the context of related work on agent programming libraries and frameworks in high-level programming languages in Sect. 7. Finally, Sect. 8 concludes the chapter by discussing limitations and future work.

2 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` function was adopted as part of the ECMA Script standard[2], a similar function (`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) [15] architecture as popularized in the MAS community by frameworks like Jason [3] (as the name *JS-son* reflects) and provides an abstraction of the BDI architecture (as well as support for other reasoning loops) as a *plug and play* dependency for a widely adopted programming language. Table 1 provides a side-by-side overview of the influence of the functional programming paradigm via Lisp's `MEMBER` function on JavaScript's `includes` function as an analogy to the influence of Jason's `(event, context, body)`-plans on JS-son's `(intention-condition, body)`-plans. To further guide the design and develop-

Table 1. Evolution of a Functional Feature from Lisp to JavaScript and Development of an Agent-oriented Feature from Jason to JS-son.

	Functional programming	Agent-oriented programming
Source technology	Lisp	Jason
Source feature,	`MEMBER` function (list)	`(event, context, body)` plans
Target technology	JavaScript	
Target feature	`includes` functor (array)	`(intention-condition, body)` plans
Library/extension	Lodash (_)	JS-son
Standard feature	`includes` (ES2016)	none

ment of JS-son, we introduce three design principles that are–in their structure, as well as in their intend to avoid unnecessary overhead on the software (agent) engineering process–influenced by the *Agile Manifesto*[4].

Usability over intellectual elegance. JS-son provides a core framework for defining agents and their reasoning loops and environments, while allowing users to stick to pure JavaScript syntax and to apply their preferred libraries and design patterns to implement agent-agnostic functionality.

Flexibility over rigor. Instead of proposing a *one-size-fit-all* reasoning loop, JS-son offers flexibility in that it supports different approaches and is intended to remain open to evolve its reasoning loop as it matures.

Extensibility over out-of-the-box power. To maintain JS-son as a concise library that can be adapted to a large variety of use cases while requiring little additional learning effort, we keep the JS-son core small and abstain from adding complex, special-purpose features, in particular if doing so imposed additional learning effort for JS-son users or required the use of third-party dependencies; *i.e.*, we maintain a lean JS-son core module that is written in

[2] https://www.ecma-international.org/ecma-262/7.0/#sec-array.prototype.includes.
[3] https://underscorejs.org/#contains.
[4] http://agilemanifesto.org/.

vanilla JavaScript (does not require dependencies). Additional functionality can be provided as modules that extend the core and are managed as separate packages.

3 Architecture and Reasoning Loops

The library provides object types for creating agent and environment objects, as well as functions for generating agent beliefs, desires, intentions, and plans[5]. The **agent** implements the BDI concepts as follows:

Beliefs: A belief can be any JavaScript Object Notation (JSON[6]) 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: A plan's *head* specifies which intention needs to be active for the plan to be pursued. The plan body specifies how the plan should update the agent's beliefs and determines the actions the agent should issue to the environment.

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* (see Rao and Georgeff [15]).
3. It runs the plans that are active according to the updated intentions, while also updating the agent beliefs (if specified in the plans).
4. It issues action requests that result from the plans to the environment.

It is also 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. Alternatively, belief-desire-plan/belief-intention-plan reasoning loops are supported; these approaches bear similarity to the belief-goal-plan approach of the *GOAL* language [8]. Figure 1a depicts the reasoning loops that are supported by standard JS-son agents.

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.

[5] The library–including detailed documentation, examples, and tests–is available at https://github.com/TimKam/JS-son.

[6] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

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.

Figure 1b depicts the environment's agent and state management process[7].

4 Implementing JS-son Agents

This section explains how to implement JS-son agents, by first giving a detailed explanation of the most important parts of the JS-son core API and then providing a programming tutorial.

4.1 JS-son Core API

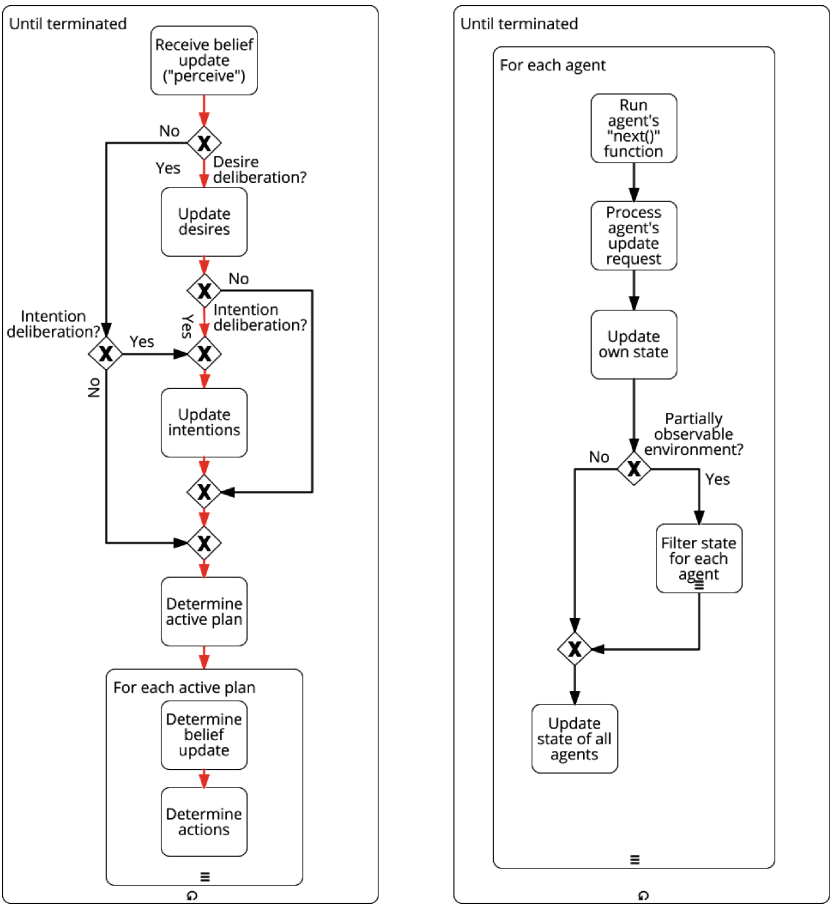
The JS-son core API provides two major abstractions: one for agents and one for environments[8]. In addition, the agent requires the instantiation of beliefs, desires, and plans. Note that intentions are generated dynamically, as is explained below.

Agents. An agent is instantiated by calling the *Agent* function with parameters that specify the agent's identifier (a text string), as well as its initial beliefs, desires, plans, and a *preference function generator*. Beliefs, desires, and plans are generated by the $Belief$, $Desire$, and $Plan$ functions, respectively. Beliefs and desires consist of an identifier (key) and a body (value). A belief body can be any valid JSON object or property (number, string, *null*, boolean, or array). A desire body is a function that processes the agent's current beliefs and returns the processing result. A plan has two functions; one as its *body* and one as its *head*. The head determines–based on an agent's beliefs–if the plan body should be executed. The body determines agent actions, as well as belief updates, taking the agent's beliefs as an optional input. *Intentions* are created by a preference function generator, a higher-order function that, based on the agents' current desires and beliefs, generates a function that reduces the agents' desires to intentions. Table 3a documents the *Agent* function signature, whereas Tables 3b, 3c, and 3d document the signatures for the $Belief$, $Desire$, and $Plan$ functions, respectively.

Environment. The environment is generated by the *Environment* function that takes as its input an array of JS-son agents, an initial state definition (JSON object), and functions for updating the environment's state, visualizing it, and pre-processing (filtering or manipulating) it before exposing the state to the agents. The *update* function processes the agents' actions; for each agent, it determines how the environment's state should be updated, based on the

[7] In its current version, JS-son executes all steps *synchronously*. Supporting the asynchronous execution, in particular of agent plans is future work, as discussed in Sect. 8.

[8] Here, we only explain the core functionality for instantiating agents and environments. A comprehensive, continuously updated documentation of the JS-son API is available at https://js-son.readthedocs.io/en/latest/.



(a) JS-son reasoning loop. The *XOR* gateways allow for different reasoning loop approaches. The red sequence flows indicate the path of the belief-desire-intention-plan reasoning loop.

(b) JS-son environment: agent and state management process. The *XOR* gateway allows for partially and fully observable environments.

Fig. 1. JS-son reasoning and environment loop.

current state, the agent's actions, and the agent's identifier. The state update is then visualized as specified by the *render* function. In case a visualization is not necessary, the default *render* function makes the environment log each iteration's state to the console. The *stateFilter* function filters or manipulates the state as perceived by a particular agent, based on this agent's identifier and its current beliefs; by default (if no *stateFilter* function is specified), the state is returned unfiltered to the agent(s). Table 3 documents the environment's function signature.

Table 2. Function signature of the JS-son *Agent* and its components.

Name	Type	Description
(a) JS-son *Agent* function signature		
id	String	Unique identifier of the agent
beliefs	Object	Initial beliefs of the agents
desires	Object	The agent's desires
plans	Array	The agent's plans
preferenceFunctionGenerator	Array	Preference function generator; by default (if no function is provided), the preference function turns all desires into intentions
Returns	Object	JS-son Agent object
(b) JS-son *Belief* function signature		
Name	**Type**	**Description**
id	String	Unique identifier of the belief
value	Any (needs to be valid JSON object or JSON value)	The belief's initial value
Returns	Object	JS-son Belief object
(c) JS-son *Desire* function signature		
id	String	Unique identifier of the belief
body	Function	Function for computing the desires value based on current beliefs
Returns	Object	JS-son Desire object
(d) JS-son *Plan* function signature		
head	Function	Determines if plan is active
body	Function	Determines the execution of actions and update of beliefs
Returns	object	Plan object

Table 3. JS-son *Environment* function signature

Name	Type	Description
agents	Array of JS-son agents	Agents that the environment is managing
state	Object	Initial state of the environment
update	Function	Processes agent actions and updates the environment's state
render	Function	Visualizes the environment's current state
stateFilter	Function	Filters/manipulates the state that agents should perceive
Returns	Object	Plan object

4.2 Tutorial

The tutorial explains how to program *belief-plan* agents using a minimal example[9]. Running the example requires the creation of a new Node.js project (`npm init`), the installation of the `js-son-agent` dependency, and the import of the `JS-son` library.

```
const {
      Belief,
      Plan,
      Agent,
      Environment } = require('js-son-agent')
```

The tutorial implements the *Jason room example*[10] with JS-son. In the example, three agents are in a room:

1. A porter that locks and unlocks the room's door if requested;
2. A paranoid agent that prefers the door to be locked and asks the porter to lock the door if this is not the case;
3. A claustrophobe agent that prefers the door to be unlocked and asks the porter to unlock the door if this is not the case.

The simulation runs twenty iterations of the scenario. In an iteration, each agent acts once. All agents start with the same beliefs. The belief with the ID `door` is assigned the object {`locked: true`}; *i.e.*, the door is locked. Also, nobody has so far requested any change in door state (`requests: []`).

```
const beliefs = {
  ...Belief('door', { locked: true }),
  ...Belief('requests', [])
}
```

[9] Tutorials that present more complex examples are available in the JS-son project documentation https://js-son.readthedocs.io.

[10] https://github.com/jason-lang/jason/tree/master/examples/room.

Now, we define the porter agent. The porter has the following plans:

1. If it does not believe the door is locked and it has received a request to lock the door (head), lock the door (body).
2. If it believes the door is locked and it has received a request to unlock the door (head), unlock the door (body).

```
const plansPorter = [
     Plan(
          beliefs =>
               !beliefs.door.locked &&
               beliefs.requests.includes('lock'),
          () => [{ door: 'lock' }]
     ),
     Plan(
          beliefs =>
               beliefs.door.locked &&
               beliefs.requests.includes('unlock'),
          () => [{ door: 'unlock' }]
     )
]
```

We instantiate a new agent with the belief set and plans. Because we are not making use of desires in this simple belief-plan scenario, we pass an empty object as the agent's desires.

```
const porter = new Agent('porter', beliefs, {}, plansPorter)
```

Next, we create the paranoid agent with the following plans:

1. If it does not belief the door is locked (head), it requests the door to be locked (body).
2. If it beliefs the door is locked (head), it broadcasts a thank you message for locking the door (body).

```
const plansParanoid = [
  Plan(
    beliefs => !beliefs.door.locked,
    () => [{ request: 'lock' }]
  ),
  Plan(
    beliefs => beliefs.door.locked,
    () => [{ announce: 'Thanks for locking the door!' }]
  )
]

const paranoid = new Agent('paranoid', beliefs, {}, plansParanoid)
```

The last agent we create is the paranoid one. It has these plans:

1. If it beliefs the door the door is locked (head), it requests the door to be unlocked (body).
2. If it does not belief the door is locked (head), it broadcasts a thank you message for unlocking the door (body).

```
const plansClaustrophobe = [
     Plan(
          beliefs => beliefs.door.locked,
          () => [{ request: 'unlock' }]
     ),
     Plan(
          beliefs => !beliefs.door.locked,
          () => [{ announce: 'Thanks for unlocking the door!' }]
     )
]

const claustrophobe = new Agent(
     'claustrophobe',
     beliefs,
     {},
     plansClaustrophobe
)
```

Now, as we have defined the agents, we need to specify the environment. First, we set the environments state, which is–in our case–consistent with the agents' beliefs.

```
const state = {
  door: { locked: true },
  requests: []
}
```

To define how the environment processes agent actions, we implement the `updateState` function. The function takes an agent's actions, as well as the agent's identifier and the current state to determine the environment's state update that is merged into the new state `state = ...state, ...stateUpdate`.

```
const updateState = (actions, agentId, currentState) => {
     const stateUpdate = {
          requests: currentState.requests
     }
     actions.forEach(action => {
          if (action.some(action => action.door === 'lock')) {
               stateUpdate.door = { locked: true }
               stateUpdate.requests = []
               console.log(`${agentId}: Lock door`)
          }
          if (action.some(action => action.door === 'unlock')) {
               stateUpdate.door = { locked: false }
               stateUpdate.requests = []
               console.log(`${agentId}: Unlock door`)
          }
          if (action.some(action => action.request === 'lock')) {
               stateUpdate.requests.push('lock')
               console.log(`${agentId}: Request: lock door`)
          }
          if (action.some(action => action.request === 'unlock')) {
               stateUpdate.requests.push('unlock')
               console.log(`${agentId}: Request: unlock door`)
          }
          if (action.some(action => action.announce)) {
               console.log(`${agentId}: ${
                    action.find(
                         action => action.announce
                    ).announce
               }`)
          }
     })
     return stateUpdate
}
```

To simulate a partially observable world, we can specify the environment's `stateFilter` function, which determines how the state update should be shared with the agents. However, in our case we simply communicate the whole state update to all agents, which is also the default behavior of the environment, if no `stateFilter` function is specified.

```
const stateFilter = state => state
```

We instantiate the environment with the specified agents, state, update function, and filter function.

```
const environment = new Environment(
     [paranoid, claustrophobe, porter],
     state,
     updateState,
     stateFilter
)
```

Finally, we run 20 iterations of the scenario.

```
environment.run(20)
```

5 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[11]-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[12] 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.

Function-as-a-Service. The term *serverless* [1] computing refers to information technology that allows application developers to deploy their code via the infrastructure and software ecosystem of third-party providers without needing to worry about the technical details of the execution environment. The provision of *serverless* computing services is often referred to as *Function-as-a-Service* (FaaS). Most FaaS providers, like Heroku[13], Amazon Web Services Lamda[14], and Google Cloud Functions[15], provide Node.js support for their service offerings and allow for the deployment of JavaScript functions with little setup overhead. Consequently, JS-son can emerge as a convenient tool to develop agents and multi-agent systems that are then deployed as *serverless* functions. For a running example, see Subsection 6.4.

6 Examples

We provide four demonstrations that show how JS-son can be applied. The code of all demonstration is available in the JS-son project repository (https://github.com/TimKam/JS-son).

[11] https://jupyter.org/.
[12] https://nodejs.org/.
[13] https://devcenter.heroku.com/articles/getting-started-with-nodejs.
[14] https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-handler.html.
[15] https://cloud.google.com/functions/docs/concepts/nodejs-8-runtime.

6.1 JS-son Meets Jupyter

The first demonstration shows how JS-son can be integrated with data science tools, *i.e.*, with Python libraries and Jupyter notebooks[16]. As a simple proof-of-concept example, we simulate opinion spread in an agent society and run an interactive data visualization. The example simulates the spread of a single boolean belief among 100 agents in environments with different *biases* regarding the facilitation of the different opinion values. Belief spread is simulated as follows:

1. The scenario starts with each agent announcing their beliefs.
2. In each iteration, the environment distributes two belief announcements to each agent. Based on these beliefs and possibly (depending on the agent type) the past announcements the agent was exposed to, each agent announces a new belief: either $true$ or $false$.

The agents are of two different agent types (*volatile* and *introspective*):

Volatile. Volatile agents only consider their current belief and the latest belief set they received from the environment when deciding which belief to announce. Volatile agents are "louder", *i.e.*, the environment is more likely to spread beliefs of volatile agents. We also add bias to the announcement spread function to favor true announcements.

Introspective. In contrast to volatile agents, introspective agents consider the past five belief sets they have received, when deciding which belief they should announce. Introspective agents are "less loud", *i.e.*, the environment is less likely to spread beliefs of volatile agents.

The agent type distribution is $50, 50$. However, 30 volatile and 20 introspective agents start with $true$ as their belief, whereas 20 volatile and 30 introspective agents start with $false$ as their belief. Figure 2a shows an excerpt of the Juypter notebook.

6.2 JS-son in the Browser

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. In this example, JS-son is fully integrated into a JavaScript build and compilation pipeline that allows writing modern, idiomatic JavaScript code based on the latest ECMAScript specification, as it compiles this code into cross-browser compatible, *minified* JavaScript. The demonstration makes use of JS-son's simplified belief-plan approach[17]. Each Game of Life *cell* is represented by an agent that has two beliefs: its own state (active or inactive) and the number of its *active* neighbors. At each simulation tick, the agent decides based on its beliefs, if it

[16] 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.

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

should register a change in its status (from active to inactive or vice versa) with the environment. After all agents have registered their new status, the environment updates the global game state accordingly and passes the new number of active neighbors to each agent. Figure 2b depicts the *Game of Life* application.

6.3 Learning JS-son Agents

The third demonstration shows how *learning* JS-son agents can be implemented in a browser-based grid world[18]. The example instantiates agents in a 20×20 field grid world *arena* with the following field types:

- *Mountain* fields that the agents cannot pass.
- *Money* fields that provide a *coin* to an agent that approaches them (the agent needs to move onto the field, but the environment will return a coin and leave the agent at its current position).
- *Repair* fields that provide damaged agents with one additional health unit when approached (again, the agent needs to move onto the field, but the environment will return a health unit and leave the agent at its current position).
- *Plain* fields that can be traversed by an agent if no other agent is present on the field. If another agent is already present, the environment will reject the move, but decrease both agents' health by 10. When an agent's health reaches (or goes below) zero, it is punished by a withdrawal of 100 coins from its stash.

The agents are trained *online* (no model is loaded/persisted) using deep Q-learning through an experimental JS-son learning extension. Figure 2c shows the agents in the grid world arena.

6.4 Serverless JS-son Agents

The fourth demonstration shows how JS-son agents can be deployed to *Function-as-a-Service* providers. It is based on the belief spread simulation as introduced in the first demonstration (see Subsect. 6.1). The multi-agent simulation is wrapped in a request handler and provided as a Node.js project that is configured to run as a *Google Cloud Function*. The request handler accepts HTTP(S) requests against the `simulate` endpoint. The request *method* (*e.g.*, `GET`, `POST`, `PUT`) is ignored by the handler. Upon receiving the request, the handler runs the simulation for the specified number of *ticks*, configuring the *bias* in the agent society as specified by the corresponding request parameter (the higher the bias, the stronger the facilitation of *true* announcements). An example request against a fictional FaaS instance could be sent using the `curl` command line tool as specified in the code snippet below.

[18] This grid world is an adaptation of an environment in which learning JS-son agents are rewarded based on a specific, *fair* game-theoretical equilibrium in a given state, as presented by Kampik and Spieker [10].

```
curl -X GET 'https://instance.faas.net/simulation/simulate?ticks=20&bias=5'
```

Figure 2d depicts the simulation in the Google Cloud Functions management user interface.

7 Related Work

Over the past two decades, a multitude of agent-oriented software engineering frameworks emerged (see, *e.g.*, Kravari and Bassiliades [11]). However, most of these frameworks do not target higher-level programming languages like Python and JavaScript. In this section, we provide a brief overview of three agent programing frameworks–*osBrain*, *JAM*, and *Eve* that are indeed written in and for these two languages. We then highlight key differences to our library.

7.1 OsBrain

osBrain[19] is a Python library for developing multi-agent systems. Although osBrain is written in a different language than JS-son, it is still relevant for the comparison because it is *i)* written in a higher level programming language of a similar generation and *ii)* somewhat actively maintained[20]. Initially developed as an automated trading software backbone, the focus of osBrain lies on the provision of an agent-oriented communication framework. No framework for the agents internal reasoning loop is provided, *i.e.* osBrain does not provide BDI support. Also, osBrain dictates the use of a specific communication protocol and library, utilizing the message queue system *ZeroMQ* [9].

7.2 JavaScript Agent Machine (JAM)

Bosse introduces the *JavaScript Agent Machine* (JAM), which is a "mobile multi-agent system[...] for the Internet-of-Things and clouds" [5].

Some of JAM's main features and properties are, according to its documentation[21]:

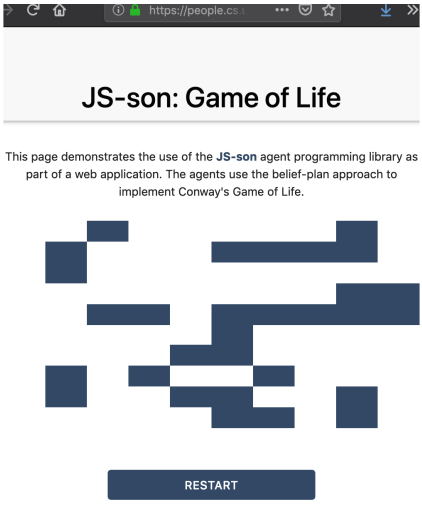
- Performance: through third-party libraries, JAM agents can be compiled to Bytecode that allows for performant execution in low-resource environments;
- Mobility and support for heterogenous environments: agent instances can be moved between physical and virtual nodes at run-time;

[19] https://osbrain.readthedocs.io/en/stable/about.html.

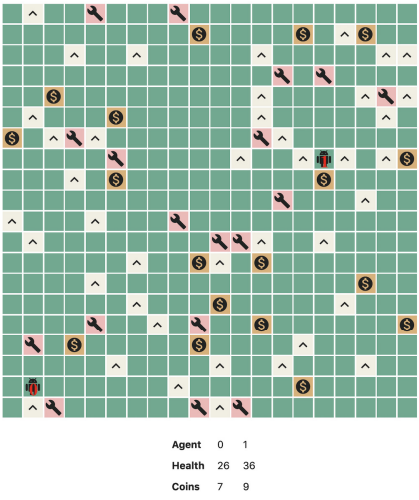
[20] As of March 2020, the last update to the source of *Eve* dates back more than 2.5 years to August 2017 (https://github.com/enmasseio/evejs/); the last update of the documentation of *JAM*–whose source code is not available–dates back more than 1.5 years to August 2018 (http://www.bsslab.de/?Software/jam). In contrast the last update of the osBrain source and documentation dates back roughly one year to April 2019 (https://github.com/opensistemas-hub/osbrain).

[21] http://www.bsslab.de/assets/agents.html.

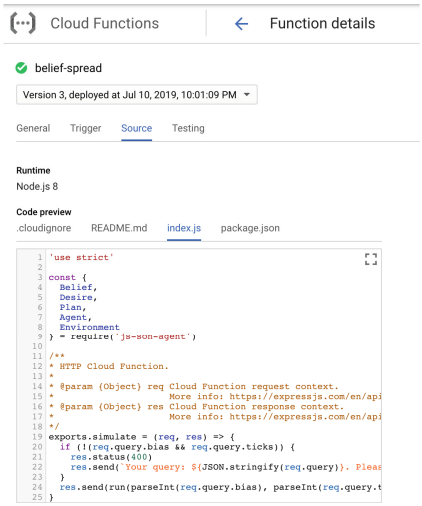
(a) Analysis of a JS-son multi-agent simulation in a Jupyter Notebook.



(b) JS-son: Conway's Game of Life, implemented as a web application.



(c) JS-son agents in a grid world.



(d) JS-son multi-agent system, deployed as a Google Cloud Function.

Fig. 2. JS-son example applications.

– Machine learning capabilities, through integration with a machine learning service platform; however, no details on how this service can be accessed are provided in the documentation.

In its initial version, JAM agents required the use of a JavaScript-like language that is syntactically not fully compliant with any standard

JavaScript/ECMAScript version [4]. However, in its latest version, it is possible to implement agent in syntactically valid JavaScript. With its focus on agent orchestration, deployment, and communications, JAM's agent internals are based on *activity-transition graphs*, which implies that its functionality overlaps little with JS-son. Another point of distinction is that the JAM source code is not openly available; instead, the JAM website[22] provides a set of installers and libraries and software development kits for different platforms that can be used as black-box dependencies.

7.3 Eve

De Jong *et al.* [7] present *Eve*, a multi-agent platform for agent discovery and communications. It is available as both a Java and a JavaScript implementation. Similar to osBrain, Eve's core functionality is an agent-oriented, unified abstraction on different communication protocols; it does not define agent internals like reasoning loops and consequently does not follow a belief-desire-intention approach. Eve is provided as Node.js package[23], but as of March 2020, the installation fails and the Node Package Manager (npm) reports 11 known security vulnerabilities upon attempted installation. Still, Eve is in regard to its technological basis similar to JS-son. With its difference in focus–on agent discovery and communications in contrast to JS-son's reasoning loops–Eve could be, if maintenance issues will be addressed, a potential integration option that a JS-son extension can provide.

7.4 Comparison - Unique JS-son Features

To summarize the comparison, we list three unique features that distinguish JS-son from the aforementioned frameworks.

Reasoning loop focus with belief-desire-intention support. Of the three frameworks, only JAM provides a dedicated way to frame the reasoning loop of implemented agents, using activity-transition graphs. Still, the core focus of *all three* libraries is on communication and orchestration, which contrasts the focus of JS-son as a library that has a reasoning loop framework at its core and aims to be largely agnostic to specific messaging and orchestration approaches.

Full integration with the modern JavaScript ecosystem. As shown in Sect. 6, JS-son fully integrates with the JavaScript ecosystem across runtime environments. This is in particular a contrast to JAM, which provides installers that obfuscate the proprietary source code and require a non-standard installation process. This can potentially hinder integration into existing software ecosystems that rely on *industry standard* approaches to dependency management for continuous integration and delivery purposes.

[22] http://www.bsslab.de/?Software/jam.
[23] https://www.npmjs.com/package/evejs.

While Eve attempts to provide an integration that allows for a convenient deployment in different environments, for example through continuous integration pipelines, it does in fact not provide a working, stable, and secure installation package.

Dependency-free and open source code. JS-son is a light-weight, open source library that does not ship any dependencies in its core version, but rather provides modules that require dependencies as *extensions*. In contrast, adopting JAM requires reliance on closed/obfuscated source code, whereas osBrain and Eve require a set of dependencies, which are in the case of Eve–as explained before–not properly managed.

8 Conclusions and Future Work

This chapter presents a lean, extensible library that provides simple abstractions for JavaScript-based agent programming, with a focus on reasoning loop specification. To further increase the library's relevance for researchers, teachers, and practitioners alike, we propose the following work:

Support a distributed environment and interfaces to other MAS frameworks. 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[24]. However, as highlighted in a previous publication [14], FIPA ACL does not support communication approaches that have emerged as best practices for real-time distributed systems like *publish-subscribe*. Also, the application of JS-son in a distributed context can benefit from the enhancement of agent-internal behavior, for example through a feature that supports the asynchronous execution of plans.

Implement a reasoning extension. To facilitate JS-son's reasoning abilities, additional JS-son extensions can be developed. From an applied perspective, integrations with business rules engines can bridge the gap to traditional enterprise software, whereas a JS-son extension for *formal argumentation* (see, *e.g.*, Bench-Capon and Dunne [2]) can be of value for the academic community.

Move towards real-world usage. To demonstrate the feasibility of JS-son, it is important 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 can facilitate real-world adoption. Still, future work needs to evaluate how useful the abstractions JS-son provides are for industry software engineers.

Implement a Python port. While JS-son can be integrated with the Python ecosystem, for example via Jupyter notebooks, doing so implies technical overhead and requires knowledge of two programming languages[25]. To facilitate

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

[25] Also, the module that allows for Node.js-Python interoperability (https://github.com/pixiedust/pixiedust_node) has some limitations, *i.e.* it lacks Python 3 support.

the use of agents in a data science and machine learning context, we propose the implementation of *Py_son*, a Python port of JS-son.

Acknowledgements. The authors thank the anonymous reviewers, as well as Cleber Jorge Amaral, Jomi Fred Hübner, Esteban Guerrero, Yazan Mualla, Amro Najjar, Helge Spieker, Michael Winikoff, and many others for useful feedback and discussions. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

1. Baldini, I., et al.: Serverless computing: current trends and open problems. In: Chaudhary, S., Somani, G., Buyya, R. (eds.) Research Advances in Cloud Computing, pp. 1–20. Springer, Singapore (2017). https://doi.org/10.1007/978-981-10-5026-8_1
2. Bench-Capon, T.J., Dunne, P.E.: Argumentation in artificial intelligence. Artif. Intell. **171**(10–15), 619–641 (2007)
3. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. Wiley Series in Agent Technology. Wiley, Chichester (2007)
4. Bosse, S.: Unified distributed computing and co-ordination in pervasive/ubiquitous networks with mobile multi-agent systems using a modular and portable agent code processing platform. Procedia Comput. Sci. **63**, 56–64 (2015)
5. Bosse, S.: Mobile multi-agent systems for the Internet-of-Things and clouds using the Javascript agent machine platform and machine learning as a service. In: 2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud), pp. 244–253. IEEE (2016)
6. Chollet, F.: Deep Learning with Python, 1st edn. Manning Publications Co., Greenwich (2017)
7. De Jong, J., Stellingwerff, L., Pazienza, G.E.: Eve: a novel open-source web-based agent platform. In: 2013 IEEE International Conference on Systems, Man, and Cybernetics, pp. 1537–1541. IEEE (2013)
8. Hindriks, K.V.: Programming rational agents in GOAL. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming, pp. 119–157. Springer, Boston, MA (2009). https://doi.org/10.1007/978-0-387-89299-3_4
9. Hintjens, P.: ZeroMQ: Messaging for Many Applications. O'Reilly Media Inc., Sebastopol (2013)
10. Kampik, T., Spieker, H.: Learning agents of bounded rationality: rewards based on fair equilibria. In: 2019 The 31st Annual Workshop of the Swedish Artificial Intelligence Society (SAIS) (2019)
11. Kravari, K., Bassiliades, N.: A survey of agent platforms. J. Artif. Soc. Soc. Simul. **18**(1), 11 (2015)
12. Logan, B.: An agent programming manifesto. Int. J. Agent-Oriented Softw. Eng. **6**(2), 187–210 (2018)
13. Mascardi, V., et al.: Engineering multi-agent systems: state of affairs and the road ahead. SIGSOFT Eng. Notes (SEN) **44**(1), 18–28 (2019)
14. Nieves, J.C., Espinoza, A., Penya, Y.K., De Mues, M.O., Pena, A.: Intelligence distribution for data processing in smart grids: a semantic approach. Eng. Appl. Artif. Intell. **26**(8), 1841–1853 (2013)

15. 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 (1991)
16. Smilkov, D., et al.: TensorFlow.js: machine learning for the web and beyond. arXiv preprint arXiv:1901.05350 (2019)