AJAN: An Engineering Framework for Semantic Web-Enabled Agents and Multi-Agent Systems

André Antakli, Akbar Kazimov, Daniel Spieldenner,
Gloria Elena Jaramillo Rojas, Ingo Zinnikus, and Matthias Klusch

{firstname.lastname}@dfki.de
German Research Center for Artificial Intelligence,
Saarland Informatics Campus, Saarbruecken, Germany

Abstract. The development of Semantic Web-enabled intelligent agents and multi-agent systems still remains a challenge due to the fact that there are hardly any agent engineering frameworks available for this purpose. To address this problem, we present AJAN, a modular framework for the engineering of agents that builds on Semantic Web standards and Behavior Tree technology. AJAN provides a web service-based execution and modeling environment in addition to an RDF-based modeling language for deliberative agents where SPARQL-extended behavior trees are used as a scripting language to define their behavior. In addition, AJAN supports the modeling of multi-agent coordination protocols while its architecture, in general, can be extended with other functional modules as plugins, data models, and communication layers as appropriate and not restricted to the Semantic Web.

Keywords: Semantic Web · Agent Engineering Framework · Multi-Agent Systems

1 Introduction

The idea of Semantic Web-enabled intelligent agents and multi-agent systems (MAS) has been around for almost as long as the idea of the Semantic Web itself, as indicated in [9]. Nonetheless, as mentioned in [8], web-based MAS has been somewhat neglected until recently. The situation has changed with the emergence of new standards such as Linked Data (LD) or Web of Things (WoT), which aim to improve the interoperability of heterogeneous environments and provide a solid foundation for building autonomous intelligent and distributed systems. For example, [9] presents agents for manufacturing using said standards, or show, as in [13], how those can be combined with machine learning.
However, these approaches are isolated solutions that only show how agents can be used in and implemented for the Semantic Web but without some agent engineering framework for this purpose. In fact, the development of semantic web-enabled agents and multi-agent systems still requires considerable manual effort from the agent engineer. Established agent engineering frameworks such as Jason, JaCaMo and JACK are simply not natively designed to interact with

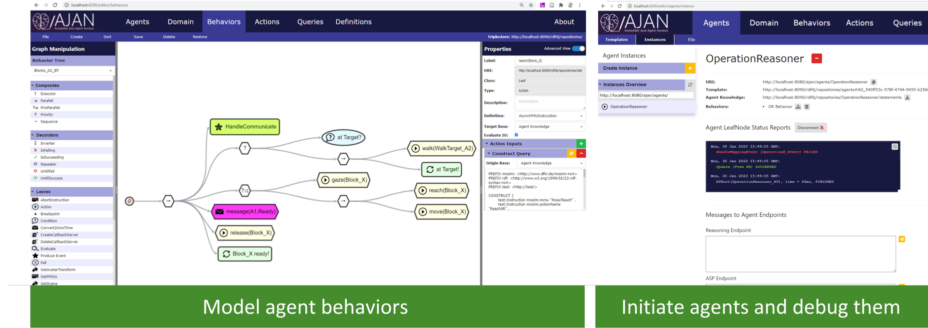


Fig. 1: The AJAN editor to model, create and debug AJAN agents

Semantic Web environments, let alone to provide an agent modeling language and environment that embeds itself homogeneously into the Semantic Web. [4] is taking a step in the right direction with a proposed OWL-based agent model language that can be combined with frameworks such as JADE. Or even [14], in which the behavior of reflexive agents can be described and executed via N3 rules. However, these approaches are insufficient in describing an agent's interaction with its environment or knowledge base, or do not even provide one. Additionally, they fail to support the engineer with templates for implementing multi-agent coordination protocols.

To address these challenges, we present AJAN, a novel, modular and extensible framework for the engineering of Semantic Web-enabled agents and multi-agent systems. AJAN relies on the SPARQL-BT paradigm to describe the event-based behavior of agents and enables them to natively interact with Semantic Web-based environments or coordinate themselves with other AJAN agents. The AJAN agent model including agent knowledge and behavior is described in RDF and SPARQL in direct support of the development of agents when they are intended for the Semantic Web. Due to its architecture, AJAN can be extended with other AI approaches but also data models as well as communication layers, allowing it to be used in various domains such as Industry 4.0, pedestrian simulation, smart living, or social services.

This paper is structured as follows. Section 2 describes the AJAN Framework. In Section 3, various application areas are presented in which AJAN is used. After discussing the related work in Section 4, we conclude the paper in Section 5.

2 The AJAN Framework

AJAN (Accessible Java Agent Nucleus) is an agent and multi-agent system framework primarily designed for the use in the Semantic Web. This framework consists of multiple RDF based languages to model agents, a RDF triplestore for data management, the AJAN-service to execute agents and the AJAN-editor to model them. In general, with AJAN, Linked Data (LD) based deliberative agents can be modeled and executed. These agents are equipped with a RDF-based agent knowledge. The behavioral model of an AJAN agent is defined through SPARQL-extended Behavior Trees, known as SPARQL-BT. With this graphical

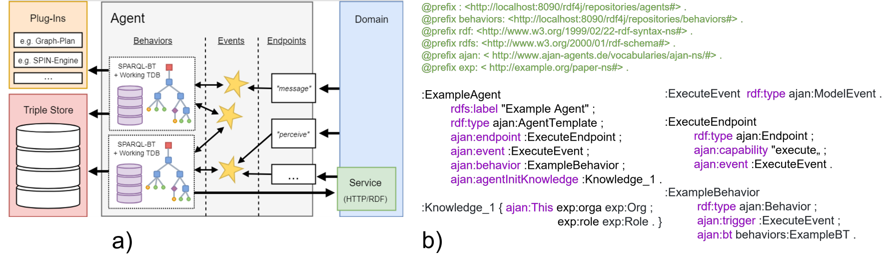


Fig. 2: a) Agent Model Overview; b) RDF example of an agent template

scripting language designed for the Semantic Web, the internal decision-making process of an agent as well as its interaction with LD environments and RDF triplestores can be realized. Due to the modular nature of the SPARQL-BT approach (see Section 2.2), it can be easily extended with additional primitives for the use in various domains or to integrate other AI approaches. One goal is not only to realize agents interacting with LD-resources, the whole AJAN system has to act as one. Therefore, with the AJAN-service, agents can be created, executed, monitored and deleted via LD-interfaces. Furthermore, the whole agent model, including SPARQL-BTs and the knowledge of individual agents, are completely defined in RDF and accessible via a RDF triplestore. Through the chosen architecture, an AJAN agent can basically interact with its own service, e.g. to create new agents, but it is also possible to give the agent a kind of 'self-awareness', since it can access its own model and monitor running behaviors at any time via SPARQL-BTs and adapt these dynamically. The AJAN-editor (see Figure 1) offers a GUI for agent and behavior modeling, which can be used at design time but also runtime, to create or delete agents and to monitor their knowledge and behaviors. A detailed description of AJAN can be found in the AJAN Wiki[1].

2.1 AJAN Agent Model

The agent model used in AJAN follows the principles of the BDI [12] paradigm, in which agents have a knowledge base and autonomously attempt to achieve agent goals using its plan library. While processing these goals, the agent can also set itself new context-based intermediate goals in order to be able to react dynamically to its agent environment. As shown in Figure 2a, an AJAN agent has a plan or behavior library in which the respective behaviors are linked to events. When an event is created, it triggers the behavior associated with it. In addition to events, an AJAN agent also has semantic goals, a subclass of events, described with pre- and postconditions. Depending on the agent's state, events and goals can be created via behaviors, thus the behavior execution can be seen as a Hierarchical Task Network. As an interface to the agent environment, an agent can have multiple LD endpoints (REST/RDF) that, when they receive data, also generate events and goals. The beliefs of an agent are stored in a RDF based agent knowledge base (KB), which can be accessed and updated through agent behaviors, implemented as SPARQL-BTs. Figure 2b shows an

[1] AJAN Agent Model Wiki: `https://github.com/aantakli/AJAN-service/wiki`

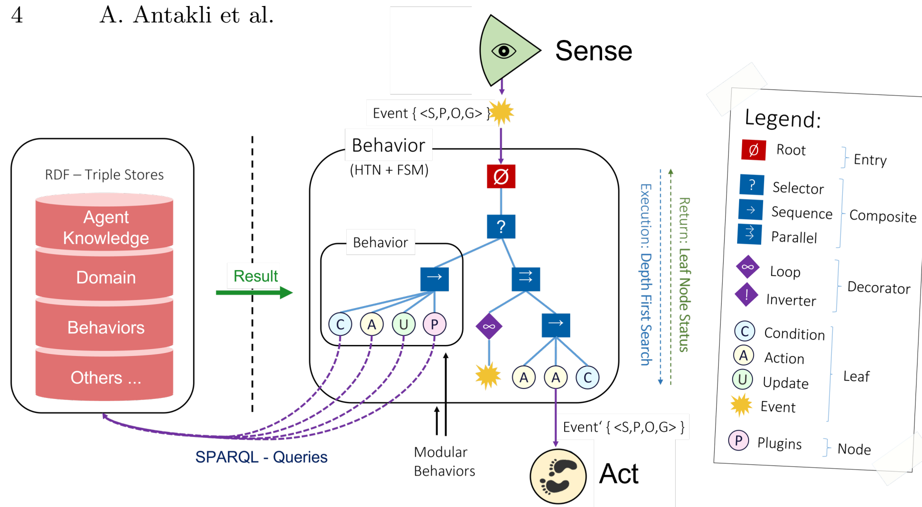


Fig. 3: SPARQL Behavior Tree Overview

example of a RDF-based agent template. An AJAN agent template can be seen as a blueprint for an AJAN agent, and thus reflects the AJAN agent model. In addition to the specification of endpoints, events or goals and behaviors, initial knowledge can also be defined. This knowledge is available to an agent from the beginning via its local knowledge base. In addition to providing domain-specific assertional knowledge, the agent model itself can be extended via the initial knowledge. For example, an agent can be assigned additional properties such as organizational affiliation or roles via the 'keyword' *ajan:This*. To model agent templates the AJAN Agent Vocabulary[2] is used. These templates are stored in an RDF triplestore as well as the local agent knowledge, terminological knowledge about the domain in which the agent acts and the plan library. Event data is also represented in RDF and is available to the triggered behaviors, via SPARQL queries. SPARQL queries (ASK queries) are also used to represent the pre- and postconditions of goals, which are validated directly on the agent knowledge.

2.2 SPARQL-BT

For modeling AJAN agent behavior, the SPARQL-BT (SBT in short) approach is used. As shown in Figure 3, SBTs are SPARQL extended Behavior Trees (BT, see [10]), for which Spieldenner et. al presented a formal specification based on the Milner Calculus in [17]. Basically, SBTs are used as an agent behavior scripting language to perform contextual SPARQL queries and to execute functionalities through nodes. In depth-first execution of these nodes, they only communicate their status such as RUNNING, SUCCEEDED or FAILED to their parent nodes. Thus, SBTs are processed like typical BTs[3] and use standard BT composite and decorator nodes, with the difference that SBTs are executed via events or goals. In general SPARQL is used in SBTs for state checking, knowledge retrieval, and manipulation. To reduce agent knowledge base requests, each

[2] AJAN agent vocabulary: `http://ajan-agents.de/vocabularies/ajan-ns`
[3] Used BT lib.: `https://github.com/libgdx/gdx-ai/wiki/Behavior-Trees`

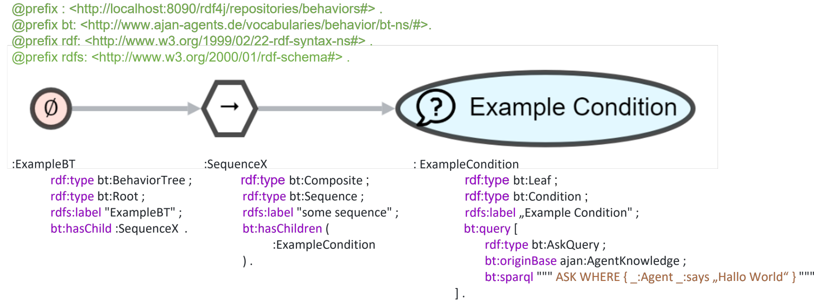


Fig. 4: Example SPARQL Behavior Tree

SBT has its own KB to store behavior specific data. To model AJAN behaviors the SPARQL-BT vocabulary[4] was defined. An example of a simple SBT, consisting of a *Root*, *Sequence* and a *Condition* node is shown in Figure 4. *:ExampleBT* is the root RDF resource of the SBT. If an AJAN *Event* appears which is triggering an initiated SBT, the child of the *Root* node is executed first. This node points to its child via *bt:hasChild*, and therefore to the RDF resource *:SequenceX* which is from type *bt:Composite* and *bt:Sequence*. If this sequence is executed, it executes its children (*bt:hasChildren*) one after the other until all children SUCCEEDED or one FAILED. In this example, its only child is from type *bt:Condition*, which executes for state checking a SPARQL ASK Query (*bt:query*) on the agent knowledge (*ajan:AgentKnowledge*). In AJAN, there are a number of other SBT nodes. For example the *Update* node, which manipulates the agent knowledge via SPARQL UPDATE; or the *Handle Event* node, which reads event and goal information via a CONSTRUCT query. Since AJAN behavior models are stored in RDF triplestores, the *Load Behavior* node can access them via a SELECT query and then initiate and execute them. Thus, an AJAN agent has the ability to dynamically generate new SBTs using *Update* nodes and subsequently execute these SBTs.

2.3 Agent Environment Interaction

There are two ways to implement an interaction between AJAN agents and their environment: **passively** via AJAN agent endpoints; or **actively** via SBT nodes:
Passive Interaction: An AJAN agent is primarily designed to interact with and as a LD resource and thus, offers multiple HTTP/RDF endpoints. In addition to a general LD endpoint to query the current agent state, including behavioral and knowledge state information, further endpoints can be defined. Such endpoints generate internal events after receiving data, which are triggering linked SBTs. Standard events are linked RDF graphs, which are available to the respective SBT for querying. To access event information within a executed SBT and store it in an agent KB, the aforementioned *Handle Event* node must be used.
Active Interaction: With the SBT *Query Domain* node a selected LD resource can be actively requested using HTTP GET. The received RDF-based data is then stored in a selected knowledge base. The SBT *Message* node can be used to

[4] SPARQL-BT vocabulary: `http://ajan-agents.de/vocabularies/behavior/bt-ns`

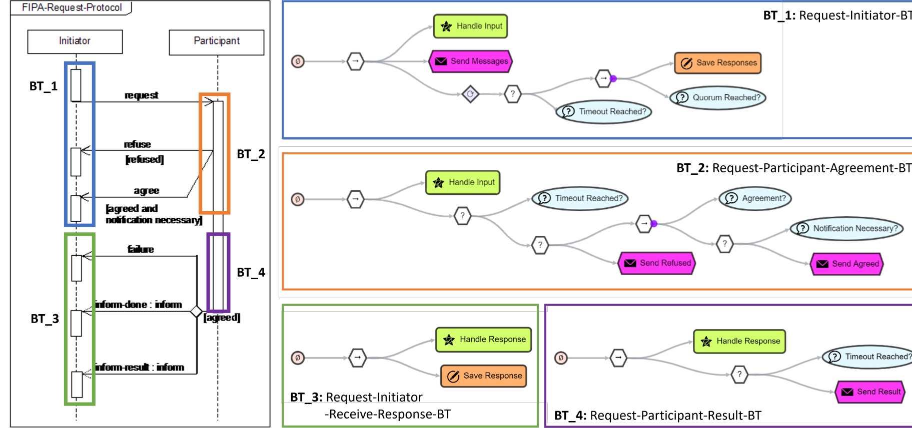


Fig. 5: FIPA Request Interaction Protocol implemented as SPARQL-BTs

configure a detailed HTTP request. Besides selecting the HTTP method to be used, additional HTTP headers can be defined, using a SPARQL SELECT query. Additionally, the *Message* node can generate a RDF-based message payload via a SPARQL CONSTRUCT query and control which data received from the requested LD resource should be stored. The *Action* node is a message node, but with pre- and postconditions like a AJAN goal. After the RDF based payload is created via CONSTRUCT, it is validated via the precondition before being sent to the LD resource, and the response is validated via the postcondition. If the validations are failing, the node status is FAILED otherwise SUCCEEDED.

2.4 Interaction Between Agents

To enable communication between agents within AJAN, the *Message* node previously described is required, as it allows to send HTTP/RDF requests to LD resources like AJAN agents. Accordingly, each AJAN agent can directly (peer-to-peer) communicate with other AJAN agents via their individual endpoints. Each agent can make use of its agent knowledge which can contain information, e.g. originating from an agents registry, about other agents in the cloud (endpoints, roles and activity status) with which they can communicate. By default, an agent has knowledge of the AJAN-service in which the agent 'lives'. With this knowledge in form of a URI, an agent can not only query its own agent model via the service's LD interface, but also other AJAN agents for communication purposes or to create or delete agents with a *Message* node.

Implementing Protocols: To ensure that agents speak the same 'language' among themselves and follow the same communication rules, the used ontologies to formulate exchanged data must not only be coordinated within the MAS the interacting agents should also use same communication protocols. In the agent context, reference must be made to FIPA, which has standardized protocols for the use in MAS to improve the interoperation of heterogeneous agents and services that they can represent [18]. For the purpose of communication of agents, these standards include e.g., speech act theory-based communicative acts or

content languages. As an example of how a FIPA protocol can be modeled in AJAN, the Request Interaction protocol is implemented using four SBTs, see Figure 5[5]. The Request-Initiator-BT (*BT_1*) sends a request to given agents via a SBT *Message* node (purple node), listens for responses via a SBT *Handle Event* node (green node), and checks for a quorum (minimum number of agents required to execute and proceed with the coordination protocol) via a SBT *Condition* node. The Request-Participant-Agreement-BT (*BT_2*) is used by the participant agent to accept or reject the request. The Request-Initiator-Receive-Response-BT (*BT_3*) receives and saves the results, and the Request-Participant-Result-BT (*BT_4*) sends the results back to the initiator agent.

2.5 Plug-In System

In order to be able to react flexibly to domain-specific circumstances and to extend the reactive planning of an agent, AJAN has a JAVA-based *Plug-In* system (see Figure 2a). This system uses several SBT and SPARQL related interface definitions with which new SBT nodes or new SPARQL functions can be implemented and integrated into AJAN. In [3] for example Answer Set Programming (ASP) is integrated to solve combinatorial problems and to extend SBTs with foresighted action sequences. In addition, classical action planning is integrated, where new SBTs are generated to achieve an RDF-described goal state. Therefore, AJAN actions and goals are translated into PDDL operators, and the agent knowledge is interpreted as the initial state. To avoid having to implement the logic of a new SBT node in JAVA, a Python interpreter is integrated as a SBT node, allowing the implementation of new SBT nodes in Python. To enable AJAN to interact not only with LD domains and thus not only via RDF messages, the *Mapping* plug-in can be used to map incoming JSON, XML or CSV information to RDF data via RML (RDF Mapping Language). To send messages with native JSON-based content, mapped from a RDF dataset, POSER [16] is used. To process telemetry data in low-bandwidth networks, MQTT (Message Queuing Telemetry Transport) SBT nodes have been integrated as well.

2.6 MAJAN: Multi-Agent Coordination Plug-In

MAJAN[6] is an extension of the agent engineering tool AJAN which provides features to implement and evaluate SPARQL-BT-based coordination of AJAN agents. As an example, we discuss how the coordination problem class Optimal Coalition Structure Generation (CSGP) can be solved with MAJAN. CSGP is defined as follows: Given a coalition game (A, v) with set A of agents, real-valued coalition values $v(C)$ for all non-empty coalitions C among agents in A, then find a partition (coalition structure) $CS*$ of A (out of all possible coalition structures CS) with maximum social welfare:

$$CS^* = argmax_{CS \in A} \sum_{m=1}^{|CS|} cv(CS_m) \tag{1}$$

[5] The presented SBTs are available under `https://github.com/AkbarKazimov/MAJAN`

[6] *MAJAN* plug-in with documentation: `https://github.com/AkbarKazimov/MAJAN`

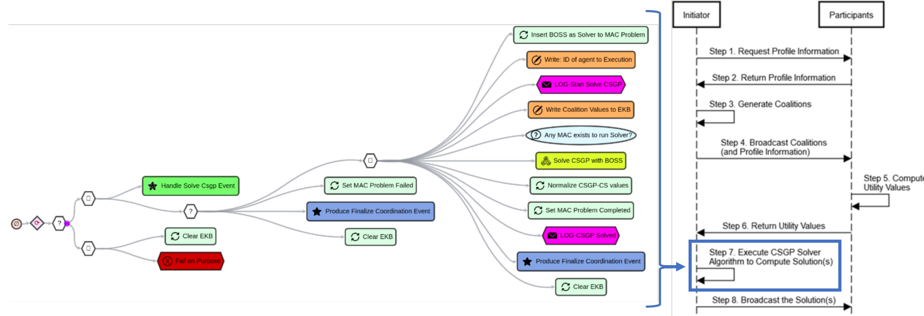


Fig. 6: Generic MAJAN MAC protocol for agent coalition structure generation

The value $v(C)$ of coalition C is the utility its members can jointly attain calculated as the sum of weighted individual utilities of coalition members, and the social welfare of a coalition structure is the sum of its coalition values. To solve CSGP problems, the BOSS algorithm [6] is used in MAJAN. As shown in Figure 6, the protocol implemented in *MAJAN* with template SPARQL-BTs for solving the clustering problem of the above-mentioned type is an appropriate adaptation of the standard FIPA-Request-Interaction protocol (see Section 2.4) for multi-agent coordination. The initiator agent of the coordination process collects profile information (e.g. individual agent properties) from all other participating agents (steps 1, 2). It then generates possible coalitions C of agents (step 3) subject to given constraints on coalition size and broadcasts them together with the collected profile information to all other participating agents (step 4). To constitute the coalition game (A, v) of the CSGP to be solved, the value of each of these coalitions must be determined. For this purpose, each agent locally computes its individual utility values for only those coalitions it is a member of and returns them to the initiator agent (steps 5, 6). Each individual utility value of an agent in some coalition is calculated only by this agent as the extent to which (the profile information of) other members of this coalition satisfy its individual preferences (that are not shared with any agent). Finally, the initiator agent computes the coalition values as the sum of relevant individual utility values collected from participating agents, calls a given CSGP solver (in *MAJAN* that is the BOSS algorithm, which is executed with the yellow-colored *CSGP* node) to compute the optimal coalition structure for the actual coalition game, and broadcasts the solution to participating agents (steps 7, 8).

2.7 AJAN Implementation

The AJAN framework consists of the AJAN-service[7] and the AJAN-editor[8], which are available on GitHub as open source software under the LGPL 2.1 and MIT licenses, respectively. Both components are executable on Windows, Linux and Mac OS. The AJAN-service is JAVA 11-based and uses Spring Boot[9] to realize it as a web service. RDFBeans[10] translates the RDF-based AJAN agent

[7] AJAN-service on GitHub: https://github.com/aantakli/AJAN-service
[8] AJAN-editor on GitHub: https://github.com/aantakli/AJAN-editor
[9] Spring Boot: `https://spring.io/`
[10] RDFBeans: `https://rdfbeans.github.io/`

model into executable JAVA code, and with PF4J[11], plug-ins are integrated into AJAN. With RDF4J[12] the interaction with RDF triplestores and data processing is realized. With it, W3C standards like OWL, SPIN (SPARQL Inferencing Notation) or SHACL (Shapes Constraint Language) can be used natively for e.g., building an agent knowledge graph with the corresponding reasoning techniques. The individual agent models and knowledge bases are stored in external triplestores which are accessible through standardized SPARQL endpoints[13]. Thus, RDF triplestores like RDF4J-server or GraphDB[14] can be used. To decrease the execution time of SBTs, and to allow that only the individual SBT has access to its own SBT knowledge base, the SBT KBs are kept in-memory RDF repositories and not externally like the agent KB. Access authorization to the AJAN-service and single triplestores, is realized using JSON Web Tokens (JWT)[15]. The AJAN-editor is based on NodeJS[16] and Ember[17].

3 Selected Applications

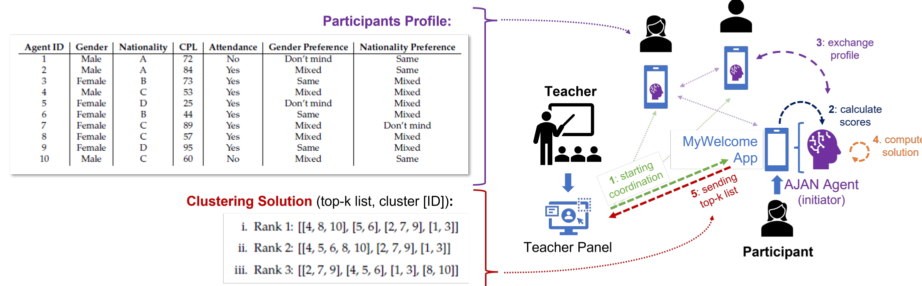
AJAN has already been used in various LD and non-LD based domains to implement agent and multi-agent systems. For example, AJAN has been used to control simulated humans, to act in a Smart Living environment or to optimize production in an Industry 4.0 context.

Human simulation. AJAN was used to control simulated entities in virtual production and traffic environments. In a human-robot collaboration scenario presented in [2], AJAN agents control simulated workers and mobile robots with LD interfaces to coordinate them for joint fulfillment of production steps. As presented in [3], AJAN was used to simulate pedestrians based on real human behavior. To realize this, an ML-based imitation model is integrated into AJAN to generate new trajectories, based on the simulated pedestrian's history and inferred targets, to steer the imitated pedestrian in a game engine.

Smart living environments. In [1], a smart living environment with AJAN agents uses the W3C Web of Things (WoT) architecture, where IoT resources have RDF-based Thing descriptions. In this scenario, AJAN agents are generating and executing new SBTs based on WoT resource descriptions to dynamically interact with these resources. For example to orchestrate them, to notify caregivers when a resident needs help.

Production optimisation. The paper [17] describes a scenario where the production within a virtual factory floor is optimized. The factory floor, production units, product plans, and available products are all represented as web resources

[11] PF4J: `https://pf4j.org/`
[12] RDF4J: `https://rdf4j.org/`
[13] SPARQL HTTP Protocol: `https://www.w3.org/TR/sparql11-http-rdf-update/`
[14] GraphDB: `https://www.ontotext.com/products/graphdb`
[15] JWT for Apache tomcat: https://github.com/andreacomo/tomcat-jwt-security
[16] NodeJS: `https://nodejs.org/en`
[17] EmberJS: `https://emberjs.com/`



Agent ID	Gender	Nationality	CPL	Attendance	Gender Preference	Nationality Preference
1	Male	A	72	No	Don't mind	Same
2	Male	A	84	Yes	Mixed	Same
3	Female	B	73	Yes	Same	Mixed
4	Male	C	53	Yes	Mixed	Mixed
5	Female	D	25	Yes	Don't mind	Mixed
6	Female	B	44	Yes	Same	Mixed
7	Female	C	89	Yes	Mixed	Don't mind
8	Female	C	57	Yes	Mixed	Mixed
9	Female	D	95	Yes	Same	Mixed
10	Male	C	60	No	Mixed	Same

Fig. 7: Language course coordination by AJAN agents

using the Linked Data Platform[18] specification. In this scenario AJAN agents represent these units and can initialize additional agents to distribute the production load. Coordination between agents is achieved through the LD environment as the communication layer, which allows agents to store RDF information that other agents can indirectly perceive. This approach is elaborated further in [15].

Language course coordination. In this application, AJAN agents help to coordinate language courses for third-country nationals (TCNs) as social services in the European project WELCOME[19]. Each TCN is registered at a mobile cloud-based MyWELCOME app with an AJAN agent as personal assistant for service provision and coordination. The agents perform semantic service selection with an appropriate OWL-S service matchmaker, and can coordinate with each other to plan optimal groupings of TCNs for each lesson of a Language Learning Course (LLC, cf. Fig. 7). The goal is to assign TCNs to working groups that optimize their language learning based on individual preferences, progress, and teacher-specified constraints, such as group size and course progress level. Such constraints can be set in a *teacher panel*, and participants can set their individual preferences in their app. The agents aim to coordinate a maximally suitable working group structure for each lesson, which can be mapped to the multi-agent coalition structure generation problem. The MAJAN plug-in of AJAN (cf. Section 2.6) was used to enable agents to solve this problem with the corresponding BT-based multi-agent coordination protocol where each agent has a protocol-specific SBT executed in an event-based distributed fashion. The multi-agent coordination is performed in two phases initiated by the teacher at the end of each course lesson: In the *Assessment* phase, the MyWELCOME app used during the lesson by participants calculates appropriate scores and reports them to the respective agent of a participant, which, in turn, stores the overall course progress level of its TCN in its agent knowledge. In the *Coordination* phase, the agents coordinate to compute an optimal working group assignment and send the top-k ranked list of participants working group structures to the teacher for approval. This application has been implemented and evaluated successfully by user partners with selected TCNs.

[18] `LinkedDataPlatform:https://www.w3.org/TR/ldp/`
[19] WELCOME Project: https://welcome-h2020.eu/

4 Related Work

Semantic Web standards have been incorporated into agents and MAS since the idea of a Semantic Web appeared [9]. For example, in [13], OWL and RDF knowledge graphs have been utilized to train machine learning-based systems to interface a learning agent with the Semantic Web. [9], on the other hand, presents a web-based MAS for manufacturing, that interacts with LD and WoT environments to derive new behaviors from the semantic environment.
However, these systems are implemented for specific applications. In general, established agent system and MAS frameworks need to be adapted for use in the Semantic Web, as they have not been developed natively for it. To address this issue, [4] introduced an OWL-based ontology to describe agents and their agent-to-agent interaction, which can be translated into JADE agents, where basic FIPA specifications can be used. Other related works present ontologies and interpreters needed to translate semantically described agents into executable code for different BDI frameworks, such as [7] for JaCaMo, [11] for Jason, and [5] for JACK. However, these approaches limit the agent engineer's ability to specify behavior using Semantic Web standards and achieve a homogeneous modeling of the agent's interaction with its environment or knowledge, such as using SPARQL. Additionally, they lack flexibility in extending the agent model, necessitating adaptations to the interpreter, framework, and ontology used. An agent engineering framework that was developed specifically for the use in the Semantic Web is presented in [14]. Here, the agent behavior and its interaction with LD environments is described and executed via N3 rules, where even HTTP messages are defined in RDF. However, this framework allows to model only reflexive agents that can act only in an LD environment.

5 Conclusion

We presented AJAN, an open-source, modular and highly extensible agent engineering framework that particularly allows for the development of semantic web-enabled agents and MAS. AJAN relies on the paradigm of event-based SPARQL Behavior Tree processing and RDF for the modeling of deliberative agents and their interaction with the environment. Moreover, AJAN offers predefined Behavior Tree templates for implementing multi-agent coordination protocols such as clustering and coalition formation. AJAN has already been used for the development of agent-based applications in various domains such as human simulation, social services and production optimization.

Acknowledgements

This work has been supported by the German Federal Ministry of Education and Research (BMBF) in MOMENTUM project (01IW22001), and the European Commission in WELCOME project (870930).

References

1. Alberternst, S., Anisimov, A., Antakli, A., Duppe, B., Hoffmann, H., Meiser, M., Muaz, M., Spieldenner, D., Zinnikus, I.: From things into clouds–and back. In:

2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid). pp. 668–675. IEEE (2021)
2. Antakli, A., Spieldenner, T., Rubinstein, D., Spieldenner, D., Herrmann, E., Sprenger, J., Zinnikus, I.: Agent-based web supported simulation of human-robot collaboration. In Proc. of the 15th Int. Conf. on Web Information Systems and Technologies (WEBIST) pp. 88–99 (2019)
3. Antakli, A., Vozniak, I., Lipp, N., Klusch, M., Müller, C.: Hail: Modular agent-based pedestrian imitation learning. In: Proc. 19th Int. Conf. on Practical Applications of Agents and Multi-Agent Systems (PAAMS). Springer (2021)
4. Bella, G., Cantone, D., Asmundo, M.N., Santamaria, D.F.: The ontology for agents, systems and integration of services: recent advancements of oasis. In: Proceedings of the 23th Workshop From Objects to Agents. pp. 1–2 (2022)
5. Challenger, M., Tezel, B.T., Alaca, O.F., Tekinerdogan, B., Kardas, G.: Development of semantic web-enabled bdi multi-agent systems using sea_ml: An electronic bartering case study. Applied Sciences **8**(5), 688 (2018)
6. Changder, N., Aknine, S., Ramchurn, S.D., Dutta, A.: Boss: A bi-directional search technique for optimal coalition structure generation with minimal overlapping. In: Proc. of the AAAI Conf. on Artificial Intelligence. vol. 35, pp. 15765–15766 (2021)
7. Charpenay, V., Zimmermann, A., Lefrançois, M., Boissier, O.: Hypermedea: A framework for web (of things) agents. In: Companion Proceedings of the Web Conference 2022. pp. 176–179 (2022)
8. Ciortea, A., Mayer, S., Gandon, F., Boissier, O., Ricci, A., Zimmermann, A.: A decade in hindsight: the missing bridge between multi-agent systems and the world wide web. In: AAMAS (2019)
9. Ciortea, A., Mayer, S., Michahelles, F.: Repurposing manufacturing lines on the fly with multi-agent systems for the web of things. In: AAMAS. pp. 813–822 (2018)
10. Colledanchise, M., Ögren, P.: Behavior trees in robotics and AI: An introduction. CRC Press (2018)
11. Demarchi, F., Santos, E.R., Silveira, R.A.: Integration between agents and remote ontologies for the use of content on the semantic web. In: ICAART. pp. 125–132 (2018)
12. Rao, A.S., Georgeff, M.P.: BDI Agents: From Theory to Practice. In: Proc. of the First Inter. Conf. on Multi-Agent Systems (ICMAS-95). pp. 312–319. AAAI (1995)
13. Sabbatini, F., Ciatto, G., Omicini, A.: Semantic web-based interoperability for intelligent agents with psyke. In: Explainable and Transparent AI and Multi-Agent Systems: 4th Inter. Workshop (EXTRAAMAS 2022). pp. 124–142. Springer (2022)
14. Schraudner, D.: Stigmergic multi-agent systems in the semantic web of things. In: The Semantic Web: ESWC 2021 Satellite Events: Virtual Event, June 6–10, 2021, Revised Selected Papers, pp. 218–229. Springer (2021)
15. Schubotz, R., Spieldenner, T., Chelli, M.: stigld: Stigmergic Coordination of Linked Data Agents. In: The 6th International Conference on Bio-inspired Computing: Theories and Applications (BIC-TA 2021) (2021)
16. Spieldenner, D.: Poser: A semantic payload lowering service. In: Proc. of the 18th Inter. Conf. on Web Information Systems and Technologies (WEBIST). SCITEPRESS (2022)
17. Spieldenner, T., Antakli, A.: Behavior trees as executable representation of milner calculus notations. In: 2022 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT). IEEE (2022)
18. Suguri, H.: A standardization effort for agent technologies: The foundation for intelligent physical agents and its activities. In: Proc. of the 32nd Annual Hawaii Int. Conf. on Systems Sciences (HICSS-32). p. 10. IEEE (1999)