AJAN: Accessible Java Agent Nucleus

André Antakli, Ingo Zinnikus, Johannes Ebersold, Wolfgang Herget, and Daniel Spieldenner

German Research Center for Artificial Intelligence (DFKI) GmbH, Saarland University Campus, 66123 Saarbruecken, Germany firstname.lastname@dfki.de
http://www.dfki.de

Abstract. The Internet of Things (IoT) is currently experiencing great interest in both industry and politics. Due to the growing number of resources and data available, the standardization efforts with regard to the Web (the application layer of the Internet) are increasing. Heterogeneous systems are therefore poised to become easier to integrate but also usable for the end-user. For this reason, the IoT is also becoming increasingly relevant to the general public where users e.g. get the possibilities to network their home and to control it according to their needs. Generally, however, the problem is simply to orchestrate individual services to a higher-quality intelligent application. In this context Multi-agent Systems (MAS) have already proven useful in the past. This classical AI paradigm allows abstracting the implementation details of the resources they represent and making large distributed systems manageable and intelligent through their autonomy and interconnectedness. We introduce the Accessible Java Agent Nucleus (AJAN), our first steps towards the vision of an easy and intuitive to use MAS for different user groups for various domains. AJAN is designed as a modular and extensible MAS framework based on standard web technologies such as REST and RDF and is implemented as a web service. By using the intuitive and extensible AI paradigm, Behavior Trees and an open JAVA interface, AJAN can be extended with additional general AI but also domain-specific technologies. Thus, a large user base is enabled to realize intelligent AI-driven applications on the Web.

Keywords: IoT · Multi-Agent Systems · Linked Data · Behavior Tree.

1 Introduction

Nowadays, the Internet is becoming an increasingly attractive place to implement applications. This is due not only to its high scalability, accessibility and expandability, but also to the growing number of services. Data, computing and storage capacity are readily available to be consumed. As a result, there is an ever faster trend to shift from strongly linked, homogeneous and closed software systems to heterogeneous and partially open architectures distributed on the Internet. According to [17], the Internet of Things (IoT) market alone is expected to grow threefold worldwide between 2014 and 2020 to around 9 trillion US-Dollars. In

addition to the large, frequent topics such as smart living and health care, it is above all the manufacturing industry, especially the automotive industry, which will increasingly rely on these IoT technologies and standard web paradigms. This is particularly evident from efforts at national level. The German Government, for example, has been driving the so-called next industrial revolution with the "Industrie 4.0" vision [6] for several years. This strategy is intended to establish highly flexible and automated production, which is to be implemented through the most comprehensive possible networking of production resources and the extensive use of artificial intelligence (AI). Today, any "thing" can probably be integrated into the world's largest application, the Web, with a little effort. Only through a common "understanding", various independent resources can be orchestrated into a reasonable application. In this context, the W3C [19] published several web standards and paradigms. On top of the Hypertext Transfer Protocol (HTTP) [39] are the Representational State Transfer (REST) [48] the architectural style of the Web, and Linked Data (LD) [10] concepts like RDF [56] or SPARQL [54]. RDF is the natural LD data model, by which relations between resources which are accessible through URIs can be described through triple statements. In turn, SPARQL queries can be applied to RDF datasets, resulting in a set of mappings as an answer. With these approaches resources can be integrated into the Web and made accessible through semantics, such like Virtual Technologies presented in [49]. With the ever-increasing number of services available and nested network structures the various data have become dispersed so much, that an abstraction layer which combines them again is required for the user to implement an application. Users can be developers with strong programming knowledge who build complex systems, but also end-users who use these systems or non-experts who e.g. want to automate their home. The multi-agent system paradigm has already proven that it can be used to realize advanced distributed applications in environments with a high diversity like the IoT or LD domains, see [20, 60, 38, 27, 31]. Individual agents of a MAS are autonomous, interconnected and to a certain extent intelligent units, which perceive their environment and decide independently how to interact with it.

The MAS paradigm is predestined to implement a higher value "intelligent" functionality of semantically described heterogeneous domains on application level, while hiding the deployment context from the user. For this reason we introduce the Accessible Java Agent Nucleus (AJAN), our first steps towards the vision of an easy and intuitive to use MAS for different user groups for various LD domains. With AJAN, the user has a modular web service at his disposal with which he can model diverse and autonomous LD based system behavior using user friendly interfaces. To ensure easy modeling of multi-agent behavior, the newly invented graphical AI-paradigm SPARQL-Behavior Tree (BT) is used and has been extended with further language constructs for semantic reasoning, and interaction in LD-domains.

This paper is structured as follows. Section 2 discusses the related work. In section 3 we then present our newly invented SPARQL-BT approach. In section 4 the AJAN system is described and is followed by an MAS scenario example in section 5. Finally, after a conclusion, we discuss the further steps in section 6.

2 Related Work

Agents of a MAS are interconnected and interact directly with resources of a domain, hiding their underlying implementation details. MAS can therefore be regarded as a middleware and have a long tradition especially in the implementation of distributed environments like IoT and LD environments, see [20, 60, 38, 27, 31]. In general, an agent is, according to [59], an autonomous software entity having sensors and actuators to perceive and interact with its environment to reach goals. An established architecture to model single agents is the Belief Desire Intention (BDI) [46] paradigm, which is implemented in frameworks such as Jadex [22] or Jason [8]. Plans are executed in a Hierarchical Task Network (HTN) [32] based manner, depending on intentions and beliefs, to reach long term goals. Other languages for agents, are Finite State Machines (FSM) [57], its modular enhancement Hierarchical FSMs (HFSM) [47], or Behavior Trees (BT) [44]. The Foundation for Intelligent Physical Agents [4] (FIPA) was established to standardize the integration of heterogeneous MAS. A well-known FIPA standards-based framework, like FIPA-OS [45], is the Java Agent Development Framework (JADE) [7]. In [29, 38, 36] JADE is used as the basis for implementing a distributed IoT system. [38] uses a RDF based Agent Programming Language (APL) extension for its agent model S-APL, allowing both, agent knowledge and behavior, to be processed in the same way. IoT systems implemented with web standards are Web of Things (WoT) [18] systems. [33, 24] present frameworks, which provide resource-oriented abstraction layers for the implementation of agent based applications in WoT environments. In [24], RDF is used to describe data fragments as well as resources themselves used by Jason agents.

One of the most important aspects of the AJAN framework is the user-friendliness. Hence, it should be possible to easily extend the agent model and the behavior models should also be usable for as large a user base as possible. In addition to graphical languages for modeling BDI agents (see [58, 21]), BTs are used more frequently in industry, e.g. for robot control [41, 43, 51]. They originate from the game industry and were primarily developed to intuitively model Non-Player Character (NPC) behavior in a modular and reusable way. There is no clearly recognized definition respectively version of this paradigm [40]. Example implementations are [3, 25, 44]. Several studies have already shown that this paradigm can be adapted to various domains and extended with further AI methods, such as reinforcement learning [26, 44] or classical planning [35, 25].

3 SPARQL-BT

For modeling agent behavior in LD domains, AJAN uses an extension of the Behavior Tree paradigm, the newly invented SPARQL-BT approach. SPARQL-BTs operate on RDF datasets and are, a combination of the BT paradigm with SPARQL. Basically, BTs are used to run context-dependent SPARQL queries. This combination allows state checking or updating and executing actions in LD-domains. Furthermore, SPARQL-BTs can be modeled in RDF, whereby a

semantic description of the behaviors they implement is available and to meet the requirements of the LD paradigm. This approach is described in section 3.2 after explaining some basics regarding Behavior Trees and SPARQL.

3.1 Background

Behavior Trees: Behavior Trees are characterized by their graphical modeling properties, making them easy to use even by non-experts. This paradigm with loops, sequences, parallels and an implicit knowledge base is often described as a combination of decision trees with state machines [40]. Typically, a BT is executed sequentially in a depth-first procedure, whereby the reactivity of the agent is realized. Goals are implicitly defined and their priority is specified by the hierarchical structure of the tree. The modularity and reusability of this paradigm results from the fact that the individual tree nodes only pass their status (RUNNING, SUCCEEDED or FAILED) to parent nodes. There are four basic node types from which a BT can be built: the root node; composite nodes; decorator nodes; and leaf nodes which interact directly with the agent environment. Decorators have only one child, the node they're decorating, whereas composites have many, but at least one child. Child nodes can again be decorators, composites or leafs like action or condition nodes. Composites decide the execution order of their child nodes. Examples are:

- **Sequence**: *Runs its children in sequence and succeeds iff all nodes succeed*
- **Selector**: *Runs its children selectively and succeeds until one node succeeds*
- **Parallel**: *Runs all children at once and succeeds until a requirement holds*

Decorator nodes are used to decide how their child node has to be executed or how its status has to be propagated up to the tree. Examples are:

- **Repeater**: *Runs its child repeatedly until a defined requirement holds*
- **Inverter**: *Inverts the resulting status of its child node*

Leaf nodes are using the agent knowledge base (KB) to execute internal (e.g. belief updates) or external actions and to check if a defined condition holds. Data, which is particularly relevant for leafs, is only exchanged within the BT via the KB. Each tree node can be seen as a stand-alone behavior, allowing higher-quality BTs to be integrated into new BTs, abstracting their complexity.

SPARQL: The official query language for processing RDF datasets has four different query forms: SELECT, is used to extract specified data as a set of variables and their bindings; CONSTRUCT, is used to return the bound variables as a valid RDF graph; ASK, returns a boolean which indicates if a graph-pattern holds; and DESCRIBE, returns all RDF triples containing in the dataset about a given resource. In a SPARQL query, the WHERE clause defines the actual searched graph-pattern, which in the simplest case contains the triples to be found with variables to be bound. For more complex patterns further clauses

are available, e.g. GRAPH to specify named RDF graphs or FILTER operations, by which e.g. unary, binary or negative restrictions can be made with. Since SPARQL 1.1 the SPARQL UPDATE [55] language is available. It can be used to manipulate the given datasets directly through INSERT and DELETE operations, where again graph-patterns within a WHERE clause define the considered triples. For more information about SPARQL, we refer to [54].

3.2 Approach Overview

SPARQL-BTs use standard BT composite and decorator nodes and are processed like typical BTs, as previously presented, but this approach defines three main new leaf node types to work on RDF-based datasets and resources using SPARQL queries. Thus, a SPARQL-BT always has one or more RDF Triple Stores that can be accessed via SPARQL endpoints.

SPARQL-BT Condition *A SPARQL-BT Condition is a BT leaf node that makes a binary statement about the presence of a graph-pattern in a RDF dataset. It returns two states after execution: SUCCEEDED and FAILED and can be used to formulate state conditions of an agent. Thereby, it performs one SPARQL 1.1 ASK query on a defined RDF dataset. The dataset can be a default graph or a named graph and is represented by its SPARQL endpoint URI. To define a SPARQL ASK query, the complete language space of the SPARQL 1.1 language with regard to ASK operations in [54] can be used.*

SPARQL-BT Update *This leaf node returns two states after execution: SUCCEEDED and FAILED and can be used to create, delete or update RDF data in a Triple Store. Thereby, it performs one SPARQL 1.1 UPDATE query on a defined RDF dataset. The dataset can be a default graph or a named graph and is represented by its SPARQL endpoint URI. To define a SPARQL UPDATE query, the complete language space of the SPARQL 1.1 UPDATE language in [55] can be used.*

SPARQL-BT Action *A SPARQL-BT Action leaf node sends a message to an external resource where an RDF dataset is sent to a defined URI endpoint via HTTP. This dataset is defined using a SPARQL 1.1 CONSTRUCT query. The complete SPARQL 1.1 language space with regard to CONSTRUCT operations in [54] can be used for this purpose. The RDF response resulting from the executed external resource is then inserted into a named graph of the knowledge base. In this context, named graphs define the source of the received result. A SPARQL-BT Action node returns three states as comparable action nodes of other BT implementations, SUCCEEDED, FAILED and RUNNING. Actions that do not immediately get a result from executed LD resources, are so-called asynchronous actions. A URI is sent to these external resources so that asynchronous actions can receive delayed results without polling and, if a result exists, switch from the RUNNING status.*

```
:ReduceTemperature_Behavior
        a bt:BehaviorTree ;
        a bt:Root ;
        rdfs:label "ReduceTemperature_Behavior" ;
        bt:hasChild
            [ a bt:Repeater ;
              bt:hasChild
                [ a bt:Sequence ;
                  bt:hasChildren (
                    [ a bt:Sequence ;
                      bt:hasChildren (
                        :GET_ShutterState
                        :GET_Radiation
                        :GET_Temperature
                        :GET_ThermostatLevel
                      )
                    ]
                    [ a bt:Sequence ;
                      bt:hasChildren (
                        :Has_Temperature
                        [ a bt:Selector ;
                          bt:hasChildren (
                            [ a bt:Sequence ;
                              bt:hasChildren (
                                :Has_ThermLevel
                                :PUT_ReduceTemp
                                :UPDATE_Protocol_TempReduce
                              )
                            ]
                            [ a bt:Sequence ;
                              bt:hasChildren (
                                :Has_RadiationHigh
                                :Has_ShutterOpen
                                :PUT_CloseShutter
                                :TurnOffLights_Behavior
                                :UPDATE_Protocol_ShutterClose
                              )
                            ]
                          )
                        ]
                      )
                    ]
                  )
                ]
            ] .
```

(a) Main SPARQL-BT

```
# ---------- SPARQL-Condition ----------

:Has_Temperature
    a bt:Condition;
    rdfs:label "Temperature > 21C°" ;
    bt:query [
        a bt:AskQuery ;
        bt:originBase "http://some/sparql/endpoint"^^xsd:anyURI ;
        # some SPARQL Ask Query
        bt:sparql """
            ASK
            WHERE  {
               GRAPH <http://some/LD-thermometer>
               {
                   ?thermometer a :Thermometer ;
                                :degreeCelsius ?temp .
               }
               FILTER (?temp > 21)
            } """^^xsd:string ;
    ] .

# ---------- SPARQL-Update ----------

:UPDATE_Protocol_TempReduce
    a bt:Update;
    rdfs:label "Protocol_TempReduce" ;
    bt:query [
        a bt:UpdateQuery ;
        bt:originBase ajan:AgentKnowledge ;
        # some SPARQL Update Query
        bt:sparql """ DELETE ... INSERT ... WHERE { ... } """^^xsd:string ;
    ] .

# ---------- SPARQL-Action ----------

:PUT_CloseShutter
    a bt:Action;
    rdfs:label "PUT: CloseShutter" ;
    bt:msgBody [
        a bt:MsgBody ;
        a bt:ConstructQuery ;
        bt:originBase ajan:AgentKnowledge ;
        # some SPARQL Update Query
        bt:sparql """ CONSTRUCT ... WHERE { ... } """^^xsd:string ;
    ] ;
    bt:serviceAction services:ShutterClose .
```

(b) SPARQL-Nodes extract

Fig. 1: Reduce-Temperature SPARQL-BT in Turtle/RDF notation

Like the data processed by a SPARQL-BT, a SPARQL-BT itself can also be declaratively modeled in RDF, for example to reason automatically about behavior capabilities. Figure 1 shows an example *:ReduceTemperature_Behavior* SPARQL-BT (see figure 3 for its graphical representation) with extracts of SPARQL-Nodes. This behavior controls smart home devices, including a weather station, a radiator thermostat, a thermometer, several lights and a controllable shutter. All devices follow the LD paradigm and are accessible via HTTP endpoints and describe their state and actions to be performed via RDF. The behavior shown in (a), which is read from top to bottom, is executed repeatedly by a repeater decorator (**bt:Repeater**). All sensor information are first read out in a sequence (**bt:Sequence**) via HTTP-GET SPARQL-BT Actions (e.g. **:GET_ShutterState**) and are thus available to the BT through named graphs. If all SUCCEEDED, a SPARQL-BT Condition (**:Has_Temperature**) is used to check the temperature. If this condition holds, the temperature has to be reduced by the following selector node (**bt:Selector**). Its first child sequence checks whether the thermostat has already been completely reduced; if so and if the sun radiation is "high", the second sub-tree closes the shutters via an HTTP-PUT SPARQL-BT Action and executes another nested behavior which turns off the lights. Both selector sub-trees use at the end a SPARQL-BT Update node to protocol the executed actions.

In this example the nodes are specified with defined types as BT nodes with the internal bt-vocabulary, e.g. **bt:Root** identifies a root node. To group child nodes, ordered RDF lists are used for composite nodes. Decorator and Roots can only have one child via **bt:hasChild**. For reusability purposes, special sub-

trees and leaf node descriptions should be referenceable, as shown in (b). Here, a SPARQL-BT Condition, a SPARQL-BT Update and a SPARQL-BT Action from *:ReduceTemperature_Behavior* are presented. Each node has a SPARQL query whose operation is specified according to its node type, e.g. **bt:AskQuery** in a SPARQL-BT Condition which queries a named graph if the internal temperature of the house is over 21°C. A SPARQL-BT Action can execute an external functionality, e.g. to close the shutters. **bt:actionService** refers to a description of this functionality and how it can be executed with an HTTP message. In the example, this description is available to the system in a triple store; it can also be referenced directly to the external functionality, using an **xsd:anyURI** (see XML Schema [53]) literal, if the desired description is provided via HTTP-GET. The message to be transmitted is determined by **bt:msgBody**.

4 AJAN Realization

4.1 Motivation and Project Background

The goal of AJAN is to develop an intuitive intelligent multi-agent system that can be easily integrated into different domains. AJAN must therefore function as a domain-neutral, modular and extensible stand-alone system and be accessible via standardized interfaces. In order to ensure user-friendliness and to be able to address as many user groups as possible, an interface must also be available with which domain-specific functionality can be implemented programmatically and as close to hardware as possible, but which is also easy to operate for non-experts. As the Web is becoming more and more attractive for the implementation of various applications, and because the W3C provides established standards for semantic domain descriptions, inference technologies, but also for system architectures and communication, the decision to develop AJAN for exactly this domain was obvious at the beginning. The AJAN framework has already been used in the ARVIDA [49] project for "intelligent" pedestrian simulation in REST/RDF domains and in the INVERSIV [61] project, for adaptively generating 3D worker instructions in distributed CPPS domains.

4.2 Architecture Overview

Figure 2 shows an abstract overview of the AJAN architecture. AJAN consists of several independent Web services: The Agent Execution Service (yellow); a plug-in system (orange); the agents domain (blue) with accessible resources (green); a Triple Store (red); and a graphical editor (violet).

RDF **Triple Stores** maintain various models and manage the knowledge of the agents. Four defined triple data bases (TDB) are available for this: the Template TDB contains different agent models, the so-called agent templates, which are used for the initialization of an agent; predefined SPARQL-BT behaviors are stored in the Behavior TDB, which is the plan library of the agent system; the Domain TDB contains the domain model with ontologies, rules but also resource descriptions; finally, a Knowledge Base (KB) TDB is available for each agent.

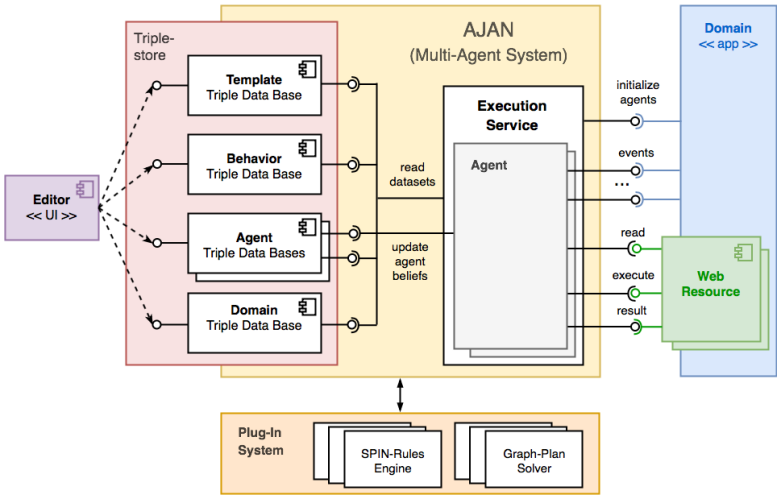


Fig. 2: Abstract AJAN architecture.

Domains, for which AJAN was primarily developed are LD environments. In these, the interaction takes place via HTTP with resources that describe their status and actions that can be performed in RDF. If available, AJAN can directly access a functional and logical description (usually using HTTP-GET) of these actions in order to execute them, but also for reasoning purposes, e.g. for automated action planning. If such information is only partially available, it can also be described in the domain model of the agent. However, AJAN provides SPARQL-BT primitives for the standard HTTP methods (GET, POST, PUT, DELETE) to interact with RDF-resources.

With the **Agent Execution Service**, the core of AJAN, agents can be created, deleted, managed and executed. Various REST interfaces are available for this purpose. For example, to create an agent, an Agent Initialization Message (AIM) must be sent to the Agent Execution Service. An AIM is an RDF dataset that contains the agent name, agent type and, if applicable, initial knowledge. The agent type is a URI that refers to an agent template in the Template TDB. This template contains the configuration of the agent with events, message endpoints but also behaviors to be used (which are described in the Behaviors TDB). SPARQL-BTs are used for the behavioral description. These can be extended via the AJAN plug-in system with further AI techniques. After initializing the agent, an agent-specific KB was created and filled with its initial knowledge.

The **AJAN Editor** is a web service, available to the user for modeling an AJAN agent. It can be used to create SPARQL-BTs, define agent templates, and edit the domain model. Figure 3 shows the AJAN editor in which the example behavior from section 3.2 is shown. A SPARQL-BT, can be created by drag-n-drop (from the left editor bar to the middle screen) SPARQL-BT primitives, e.g. SPARQL-BT Conditions (blue leaf nodes), SPARQL-BT Updates (green leaf nodes), SPARQL-BT Actions (yellow leaf nodes), but also predefined SPARQL-

BTs (white leaf node); that are then linked together. The properties of these nodes can be edited in the context bar at the right. In this example, the properties of the selected SPARQ-BT Condition (*"Temperature > 21° C"*) which checks whether the temperature is higher than 21°C can be edited. Other nodes are a repeater node (↻), sequence nodes (→) and a selector node (?).

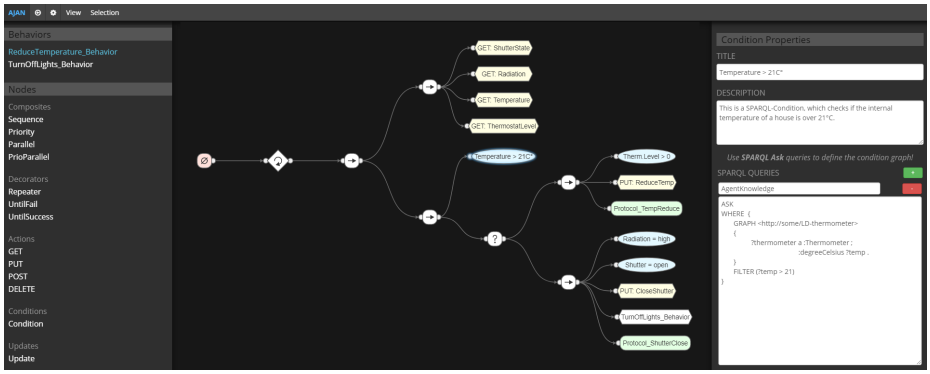
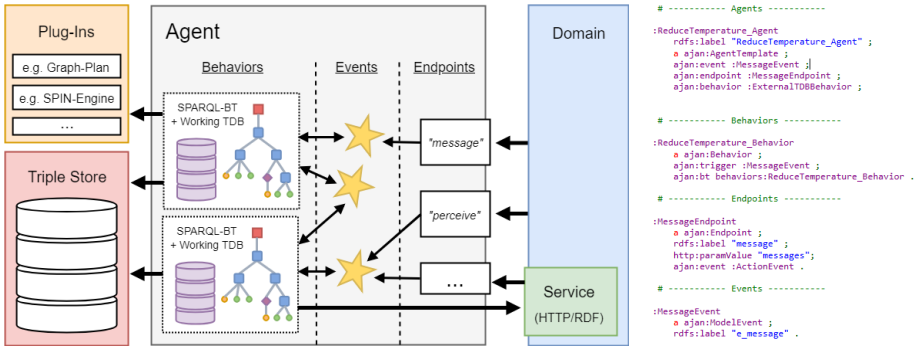


Fig. 3: AJAN Web-Editor.

4.3 Agent Model

Figure 4 shows (a) the AJAN agent model with its interactions and (b) an example agent template definition in RDF. An AJAN agent has one or more behaviors, consisting of a SPARQL-BT and its local RDF database; one or more events, each holding RDF data in the form of named graphs for behaviors; and one or more HTTP endpoints. These endpoints are the agent's interfaces to its domain and forward incoming RDF messages as named graphs in form of events. Behaviors can be linked to these events. If an event occurs, the behaviors linked to it are executed. While executing a SPARQL-BT, it can access special incoming event data by querying the events named graph. Each Behavior can also create events to trigger other behaviors. Specific event nodes for SPARQL-BTs were implemented for this purpose. Beside of these and the SPARQL-BT nodes presented in section 3.2, further nodes were realized such as a SPARQL-based composite node to dynamically choose and execute SPARQL-BTs.

By using the AJAN plug-in system, AI methods can be integrated into the system. By now AJAN has two plug-ins for: GraphPlan-based action planning in STRIPS or PDDL domains, see [42]; and a rule based inferencing engine for deductive reasoning. For this, rules are defined using the SPIN-rule language [15] based on SPARQL-construct queries and an appropriate SPIN-engine. Both plug-ins are available as SPARQL-BT nodes for behavioral modeling. When such an extended SPARQL-BT is executed, the data required for the plug-in is converted from RDF to the required format. As a result, for example, a SPARQL-BT can be synthesized during action planning or, in the case of the rule engine, the knowledge base of the agent can be extended. As already mentioned, an agent has, in addition to the working triple databases for each SPARQL-BT, one triple



(a) Overview (b) Example

Fig. 4: Agent model (a) with an example agent template (b) in RDF

store for cross-behavioral knowledge. Each SPARQL-BT can query or change the agent status, but can also use other internal or external TDBs to access the domain model using the W3C standardized SPARQL protocol [52]. Due to the strict separation of general agent knowledge and the working knowledge of the respective behaviors, the administration of runtime data becomes usable.

In (b) an example agent template is shown, which uses the **:ReduceTemperature_Behavior** presented in section 3.2). This template defines an agent (**:ReduceTemperature_Agent**) with one endpoint (**:MessageEndpoint**), one event (**:MessageEvent**) and one behavior (**:ReduceTemperature_Behavior**). By sending HTTP/RDF messages to this endpoint the defined event occurs. The specified behavior is than trigged by this event, which executes the referred **behaviors:ReduceTemperature_Behavior** stored in the behaviors TDB.

4.4 Implementation

The Triple Store was realized with RDF4J [2], an open source RDF framework based on JAVA. The upcoming RDF4J server, a Triple Store Web Service, features W3C standard SPARQL endpoints in addition to HTTP interfaces for accessing raw triples. The Agent Execution Service is based on JAVA Spring-Boot [16] and uses besides RDF4J, the framework RDFBeans [14] for the translation of RDF datasets to JAVA Beans. The gdx.ai [5] library for executing agent BTs. Communication between the Triple Store and the Agent Execution Service takes place exclusively via the W3C standardized SPARQL endpoints. The plug-in system is implemented with the former mentioned RDFBeans and the JAVA framework PF4J [13], by which ordinary JAVA classes, which implement various functionality, can be annotated as new defined SPARQL-BT nodes and integrated as project archives into AJAN. To realize the two plug-ins, the GraphPlan library JavaGP [9] was used for classic planning and RDF4J was used for integration of the SPIN engine. Finally, the editor uses the BT editor Behavior3.js [1] based on JavaScript and has been extended for modeling agents, SPARQL-BTs and domain models.

5 Example Scenario

In the previous sections, various features of AJAN have been introduced using a home automation example, in which a SPARQL-BT is executed by one AJAN agent to reduce the room temperature. In the following shop-floor scenario shown in figure 5, a multi-agent environment in the area of human-robot interaction is considered in which a worker has to carry out a production process with the help of two robots. This scenario consists of four agents. The first agent adapts the user interface of an application to the users needs, in which a worker receives context-dependent process information and may use it to send instructions to robots. The second agent plans possible actions as a worker support, to overcome critical situations and visualizes the generated action sequence via a virtual 3D avatar. The other two agents each control a robot that collaborates with the worker within a production step and supports him. Furthermore, sensors are available to monitor the production process. All entities, i.e. agents, robots, UI, simulation and sensors are represented within the LD environment (blue cloud) in the form of HTTP/RDF resources, which hides their implementation context and makes them uniformly available. Here we refer to [34] and [50], in which approaches to semantic resources in LD domains are presented.

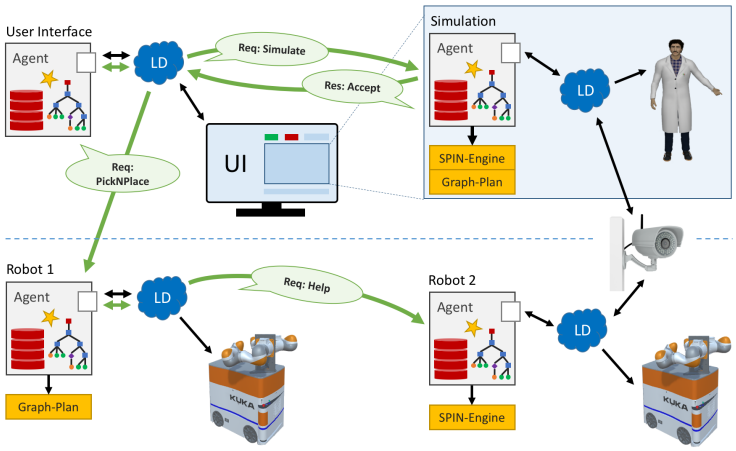


Fig. 5: Example Multi-Agent Human-Robot AJAN Scenario

An AJAN agent is able to act on this heterogeneous and distributed environment to process the RDF representation of the domain model. Hence, e.g. the simulation agent can access the sensors in order to decide autonomously how the simulated worker should be controlled depending on their states; or a robot can communicate with a UI via AJAN agents without the need to implement a direct connection. With regard to communication, various protocols can be implemented flexibly via SPARQL-BTs, in which the exchanged data describe themselves via RDF and the vocabularies or ontologies used can be "freely" accessed in a LD domain. If the simulation agent is to perform a communication

activity in reverse order, it is sufficient to reposition the corresponding graphical nodes in its SPARQL-BT. The BT paradigm allows BTs to be reused, which benefits the AJAN end-user, as the abstraction level can be customized to his needs. Using the AJAN plug-in system, AI techniques can be integrated and used graphically by the user in the form of SPARQL-BT nodes to model behaviors, e.g. that an agent steered robot can infer about its current state or to synthesize new behaviors to deal with unknown situations. For example, if *Robot 1* has the goal to transport an object to an unexpectedly blocked location, it can dynamically plan alternative actions. Due to the modular, graphical and declarative features of SPARQL-BTs, AJAN behaviors can be extended and reconfigured intuitively and dynamically as needed.

6 Conclusion and Outlook

This paper introduced AJAN, a multi-agent framework for implementing intelligent distributed applications in LD domains with the focus on user-friendliness. With the SPARQL-BT approach we also presented a novel declarative execution language for SPARQL queries. It is extendable with other AI methods and user-friendly due to its graphical properties. We have presented distributed use cases in which AJAN can be used, such as Human-Robot environments in which web standards [12] and BTs [43] already have to be retained. AJAN is designed as a control and orchestration web service and is therefore open to several different intelligent applications. Hence, unlike other agent frameworks like [22, 8, 3], behaviors can be dynamically manipulated without a compilation step and new technologies can be used via plug-ins or by direct interaction using LD. Furthermore, compared to other approaches such as [11, 24] in which knowledge-based reactive systems for interaction with WoT resources were presented, AJAN has an intuitive graphical interface that allows both application developers and end users to use the same behavior modeling and execution tool. However, to increase the usability of AJAN for non-experts even further, future efforts will focus on intuitive interaction with RDF-based data, e.g. for modeling agent domain knowledge and SPARQL queries. There are already approaches in this area that try to make RDF graphically accessible (see [30, 23]), and that want to simplify the synthesis of SPARQL queries using language technologies (see [28, 37]). AJAN has already been used to control intelligent simulations in LD environments (see [49, 61]). Nevertheless, currently there are efforts to evaluate AJAN in other distributed areas such as home automation, autonomous driving or human-robot collaboration and to integrate further planning and learning AI technologies in order to implement highly adaptive distributed systems.

Acknowledgment

The work described in this paper has been partially funded by the German Federal Ministry of Education and Research (BMBF) through the projects HybriT under the grant 01IS16026A, and REACT under the grant 01/W17003.

References

1. behavior3editor: An online visual editor for Behavior3, https://github.com/behavior3/behavior3editor, last visit: 2018-05-09
2. Eclipse RDF4j, http://rdf4j.org/, last visit: 2018-05-09
3. Epic Games: Behavior Trees, https://docs.unrealengine.com/en-us/Engine/AI/BehaviorTrees, last visit: 2018-05-09
4. FIPA: Foundation for Intelligent Physical Agents, http://www.fipa.org/, last visit: 2018-05-09
5. gdx-ai: Artificial Intelligence framework for games based on libGDX, https://github.com/libgdx/gdx-ai, last visit: 2018-05-09
6. Industrie 4.0 Plattform, https://www.plattform-i40.de, last visit: 2018-05-09
7. Jade: Java Agent DEvelopment Framework, http://jade.tilab.com/, last visit: 2018-05-09
8. Jason: a Java-based interpreter for an extended version of AgentSpeak, http://jason.sourceforge.net/wp/, last visit: 2018-05-09
9. javaGP: Graphplan implementation in Java, https://github.com/pucrs-automated-planning/javagp, last visit: 2018-05-09
10. Linked Data - W3c, https://www.w3.org/standards/semanticweb/data, last visit: 2018-05-12
11. Linked Data-Fu, http://linked-data-fu.github.io/, last visit: 2018-05-09
12. MiR - REST api http://www.mobile-industrial-robots.com/media/1479/mir-rest-api-reference-guide-100.pdf
13. PF4j: Plugin Framework for Java (PF4j), https://github.com/pf4j/pf4j, last visit: 2018-05-09
14. RDFBeans, https://rdfbeans.github.io/quickstart.html, last visit: 2018-05-09
15. SPIN - SPARQL Inferencing Notation, http://spinrdf.org/, last visit: 2018-05-09
16. Spring Boot, https://projects.spring.io/spring-boot/, last visit: 2018-05-09
17. Statista: IoT market 2020, https://www.statista.com/statistics/512673/worldwide-internet-of-things-market/, last visit: 2018-05-09
18. Web of Things at W3c, https://www.w3.org/WoT/, last visit: 2018-05-09
19. World Wide Web Consortium (W3c), https://www.w3.org/, last visit: 2018-05-09
20. 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 (Aug 2016). https://doi.org/10.1109/FiCloud.2016.43
21. Braubach, L., Pokahr, A., Jander, K., Lamersdorf, W., Burmeister, B.: Go4flex: Goal-Oriented Process Modelling. In: Essaaidi, M., Malgeri, M., Badica, C. (eds.) Intelligent Distributed Computing IV, vol. 315, pp. 77–87 (2010)
22. Braubach, L., Pokahr, A., Lamersdorf, W.: Jadex: A BDI-Agent System Combining Middleware and Reasoning. In: Unland, R., Calisti, M., Klusch, M. (eds.) Software Agent-Based Applications, Platforms and Development Kits, pp. 143–168. Birkhuser-Verlag, Basel (2005)
23. Chawuthai, R., Takeda, H.: RDF Graph Visualization by Interpreting Linked Data as Knowledge. pp. 23–39 (Mar 2016)
24. Ciortea, A., Boissier, O., Ricci, A.: Beyond Physical Mashups: Autonomous Systems for the Web of Things p. 5 (2017)
25. Colledanchise, M., Murray, R.M., Ogren, P.: Synthesis of correct-by-construction behavior trees. pp. 6039–6046. IEEE (Sep 2017), http://ieeexplore.ieee.org/document/8206502/

26. Dey, R., Child, C.: QL-BT: Enhancing behaviour tree design and implementation with Q-learning. pp. 1–8. IEEE (Aug 2013)
27. Diaconescu, I.M., Wagner, G.: Modeling and Simulation of Web-of-Things Systems as Multi-Agent Systems. In: Mller, J.P., Ketter, W., Kaminka, G., Wagner, G., Bulling, N. (eds.) Multiagent System Technologies, vol. 9433, pp. 137–153. Springer International Publishing, Cham (2015)
28. Dubey, M., Dasgupta, S., Sharma, A., Hffner, K., Lehmann, J.: AskNow: A Framework for Natural Language Query Formalization in SPARQL. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) The Semantic Web. Latest Advances and New Domains, vol. 9678, pp. 300–316. Springer International Publishing, Cham (2016)
29. Fortino, G., Guerrieri, A., Russo, W.: Agent-oriented smart objects development. In: Proceedings of the 2012 IEEE 16th International Conference on Computer Supported Cooperative Work in Design (CSCWD). pp. 907–912 (May 2012). https://doi.org/10.1109/CSCWD.2012.6221929
30. Gallego, M.A., Fernndez, J.D., Martnez-Prieto, M.A.: RDF Visualization using a Three-Dimensional Adjacency Matrix p. 5 (2011)
31. Garcia-Sanchez, F., Fernndez-Breis, J.T., Valencia-Garca, R., Gmez, J.M., Martnez-Bjar, R.: Combining Semantic Web technologies with Multi-Agent Systems for integrated access to biological resources. Journal of Biomedical Informatics **41**(5), 848–859 (Oct 2008). https://doi.org/10.1016/j.jbi.2008.05.007
32. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Elsevier (May 2004)
33. Gouach, A., Bergeret, M.: REST-A: An Agent Virtual Machine Based on REST Framework. In: Demazeau, Y., Dignum, F., Corchado, J.M., Prez, J.B. (eds.) Advances in Practical Applications of Agents and Multiagent Systems. pp. 103–112. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
34. Harth, A., Kfer, T., Keppmann, F.L., Rubinstein, D., Schubotz, R., Vogelgesang, C.: Industrielle VT-Anwendungen auf Basis von Web-Technologien Industrial VR/AR Applications based on Web Technologies p. 6 (2016)
35. Hilburn, D.: Simulating Behavior Trees A Behavior Tree/Planner Hybrid Approach. In: Game AI Pro: Collected Wisdom of Game AI Professionals. CRC Press (Sep 2013)
36. Jeon, P.B., Kim, J., Lee, S., Lee, C., Baik, D.K.: Semantic Negotiation-Based Service Framework in an M2m Environment. In: 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology. vol. 2, pp. 337–340 (Aug 2011). https://doi.org/10.1109/WI-IAT.2011.88
37. Kaufmann, E., Bernstein, A.: Evaluating the usability of natural language query languages and interfaces to Semantic Web knowledge bases. Web Semantics: Science, Services and Agents on the World Wide Web **8**(4), 377–393 (Nov 2010)
38. Khriyenko, O., Nagy, M.: Semantic Web-driven Agent-based Ecosystem for Linked Data and Services (Sep 2011)
39. Leach, P.J., Berners-Lee, T., Mogul, J.C., Masinter, L., Fielding, R.T., Gettys, J.: Hypertext Transfer Protocol – HTTP/1.1, https://tools.ietf.org/html/rfc2616, last visit: 2018-05-09
40. Martens, C., Butler, E., Osborn, J.C.: A Resourceful Reframing of Behavior Trees (Mar 2018), http://arxiv.org/abs/1803.09099
41. Marzinotto, A., Colledanchise, M., Smith, C., Ogren, P.: Towards a unified behavior trees framework for robot control. pp. 5420–5427. IEEE (May 2014)
42. Meneguzzi, F., Zorzo, A., Da Costa M Ora, M.: Propositional Planning in BDI Agents (Sep 2004). https://doi.org/10.1145/967900.967916

43. Nguyen, H., Ciocarlie, M., Hsiao, K., Kemp, C.C.: ROS commander (ROSCo): Behavior creation for home robots. In: 2013 IEEE International Conference on Robotics and Automation. pp. 467–474 (May 2013)
44. Pereira, R.d.P., Engel, P.M.: A Framework for Constrained and Adaptive Behavior-Based Agents (Jun 2015), http://arxiv.org/abs/1506.02312
45. Poslad, S., Buckle, P., Hadingham, R.: The FIPA-OS agent platform: Open Source for Open Standards p. 14 (2000)
46. Rao, A.S., Georgeff, M.P.: BDI Agents: From Theory to Practice. In: In Proceedings of the First International Conference on Multi-Agent Systems (icmas-95. pp. 312–319 (1995)
47. Risler, M.: Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines. tuprints, Darmstadt (Feb 2010)
48. Rodriguez, A.: Restful web services: The basics. IBM developerWorks p. 33 (2008)
49. Schreiber, W., Zrl, K., Zimmermann, P. (eds.): Web-basierte Anwendungen Virtueller Techniken: Das ARVIDA-Projekt Dienste-basierte Software-Architektur und Anwendungsszenarien fr die Industrie. Springer Vieweg (2017)
50. Schubotz, R., Vogelgesang, C., Antakli, A., Rubinstein, D., Spieldenner, T.: Requirements and Specifications for Robots, Linked Data and all the REST. In: Proceedings of Workshop on Linked Data in Robotics and Industry 4.0. Workshop on Linked Data in Robotics and Industry 4.0 (LIDARI-2017), 2nd Workshop on Linked Data in Robotics and Industry 4.0, located at Semantics 2017, Amsterdam, Netherlands. CEUR (2017)
51. Vouros, G.A.: Learning Conventions via Social Reinforcement Learning in Complex and Open Settings. In: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems. pp. 455–463. AAMAS '17, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2017)
52. W3C: SPARQL 1.1 Protocol (2008), https://www.w3.org/TR/sparql11-protocol/, last visit: 2018-05-09
53. W3C: W3c XML Schema Definition Language (XSD) 1.1 Part 1 (2012), https://www.w3.org/TR/xmlschema11-1/, last visit: 2018-05-09
54. W3C: SPARQL 1.1 Query Language (2013), https://www.w3.org/TR/2013/REC-sparql11-query-20130321/, last visit: 2018-05-09
55. W3C: SPARQL 1.1 Update (2013), https://www.w3.org/TR/sparql11-update/, last visit: 2018-05-09
56. W3C: RDF 1.1 Primer (2014), https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/, last visit: 2018-05-09
57. Wagner, F., Schmuki, R., Wagner, T., Wolstenholme, P.: Modeling Software with Finite State Machines: A Practical Approach. CRC Press (May 2006)
58. Warwas, S.: The Bochica Framework for Model-Driven Agent-Oriented Software Engineering. In: Agents and Artificial Intelligence. pp. 158–172. Communications in Computer and Information Science, Springer, Berlin, Heidelberg (Feb 2012)
59. Wooldridge, M., Jennings, N.R.: Intelligent Agents: Theory and Practice. Knowledge Engineering Review **10**, 115–152 (1995)
60. Xu, X., Bessis, N., Cao, J.: An Autonomic Agent Trust Model for IoT systems. Procedia Computer Science **21**, 107–113 (Jan 2013)
61. Zinnikus, I., Antakli, A., Kapahnke, P., Klusch, M., Krauss, C., Nonnengart, A., Slusallek, P.: Integrated Semantic Fault Analysis and Worker Support for Cyber-Physical Production Systems. In: 2017 IEEE 19th Conference on Business Informatics (CBI). vol. 01, pp. 207–216 (Jul 2017). https://doi.org/10.1109/CBI.2017.54