Agent Programming Language Review Assignment

Jadex

Julia Filipczak (18310726)

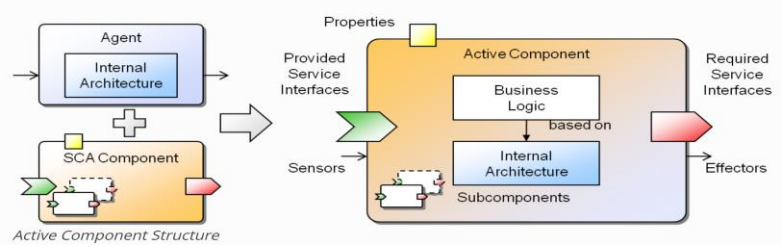
Overview

Jadex is a software framework which allows creation of goal oriented agents following the belief-desire-intention (BDI) model. It aims to make the development of agent based systems as easy as possible without sacrificing the expressive power of the agent paradigm. The objective is to build up a rational agent layer that sits on top of a middleware agent infrastructure and allows for intelligent agent construction using sound software engineering foundations.[1]

Jadex fosters the transition from traditional distributed systems to the development of multi-agent systems, object-oriented concepts as well as technologies like Java and XML. It also tries to improve upon the limitation imposed by the traditional BDI systems through the introduction of explicit goals thus allowing for the existence of goal deliberation mechanisms. Furthermore it also facilitates application development by making results from goal oriented analysis and design easily transferable to the implementation layer.[1]

The Jadex research project is conducted by the Distributed Systems and Information Systems Group at the University of Hamburg.[2] It is open source, and has been all along, although it was only made available on Github in 2018. Before being able to use Jadex for development of applications, one needs to have a java environment installed on their machine. The current version of Jadex requires minimum of version 8 and up to version 11. Other required 3rd party jars are already included within the respective distributions. The links for download of files for currently rolling release can be found at the active components download page. Alternatively, it is also possible to use Gradle or Apache Maven in which case there is no need to download the whole Jadex distribution. Upon addition of the dependency the required Jadex artifacts will be downloaded automatically.

The Jadex framework provides programming and execution facilities for distributed and concurrent systems. It is very similar to the Service Component Architecture (SCA) approach in the sense that it considers systems to be composed of components acting as service providers and consumers which are extended with agent-oriented concepts. However, in contrast to SCA, components are always active entities i.e. they posses autonomy with respect to what they do and when they perform actions. This in turn makes them analogous to agents, although they do in fact differ in the communication aspect since in the case of components communication is preferably done using service invocations.



Active Component Structure

Agent Architecture Design

The agent abstraction in the Jadex framework is based on the Belief Desire Intention (BDI) model. As its name suggests, it is characterized by 3 concepts: Beliefs – signifying the agent's knowledge about the world, Desires – representing the objectives that the agent is meant to accomplish and Intentions – the courses of actions which are currently being executed by the agent in order to achieve its desires. One of the big successes of this model is its ability to simply reduce the concept of explanation for complex human behaviour into a motivational stance. Since the causes for actions in this model are only related to desires, and ignore any other facets of cognition such as emotions, it proves quite convenient for representing the desired agent behaviours.[1] In fact this intuitive behaviour model can act as a blueprint (pattern) for many commonly found problems in agent systems thus reducing the need for manually coding aspects of the agent behaviour.[3]

Broadly speaking, an agent viewed from the outside is essentially a black box which receives and sends messages. The incoming messages as well as some internal events or goals act as input for the agent's internal reaction and deliberation mechanisms. Those mechanisms dispatch these events to already running plans or else are used to instantiate new plans from the plan library. Running plans can access and modify the belief base, send messages to other agents or cause other internal events like creating new top-level subgoals. Since the reaction and deliberation mechanisms are generally the same for all agents, it is the beliefs, goals and plans of an agent that determine its behaviour. [4] A more detailed explanation of each of the concepts mentioned is provided below.

By specifying beliefs we are providing input for the reasoning engine. During this process we can specifically address certain belief states like preconditions for plans or creation conditions for goals. The engine then checks the beliefs for relevant changes and automatically updates the goals and plans accordingly.

Goals in Jadex are viewed as concrete and transient desires of an agent. An agent will proceed to engage into appropriate actions until it drives the goal through to completion, reaches the conclusion that it's not reachable or determines that it's not desired anymore. Unlike other systems, Jadex can distinguish between goals that have just been adopted and goals that are being actively pursued through the introduction of the concept of goal lifecycle, which consists of three goal states: *option*, *active* and *suspended*. Upon adoption of a goal, it becomes an option that is added to the agent's desire structure. The deliberation mechanism is then responsible for managing the state transitions for each of the adopted goals.

There are four types of goals supported in Jadex, each playing a different role in the goal lifecycle. The *perform* goal is tied with the execution of actions and will be considered reached if the actions have been executed, regardless of their outcome. The *achieve* goal defines the desired outcome, however it is important to note that it does not provide any instructions on how to reach it since agents should be able to use several different plans in order to reach this goal. Similarly to the achieve goal, there also exists the *query* goal, which once again is not defined as some state of the world but rather a piece of information that the agent would like to know about. The fourth type of goal is called *maintain* and it involves the agent keeping track of the desired state. This can be done through continually

selecting appropriate plans for execution which will help to re-established the maintained state.

Events such as receival of a message or activation of a goal are handled through the selection of a suitable plan. For this purpose, Jadex uses the plan library. Each plan has a planning head which defines the circumstances under which the plan should be selected, this can include things like goals and events, and a plan body specifying the actions to be executed.[4] This reactive planning approach is based on the means-end reasoning process found in earlier PRS systems.[3]

The following diagrams illustrate the necessary elements and flow of processing associated with agent behaviour as described above. [4][1]

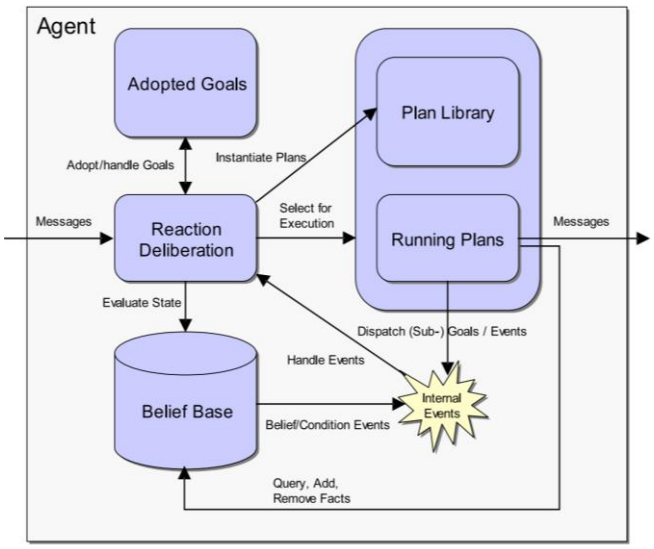


Fig. 1. Jadex abstract architecture

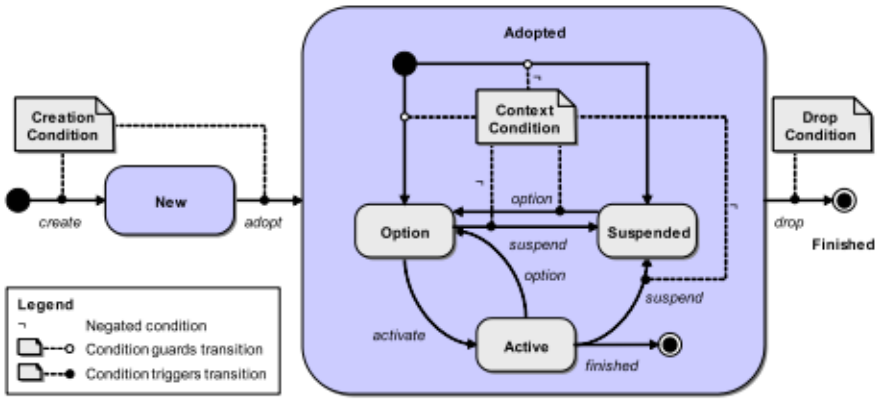


Figure 1.2. Goal lifecycle

How to Code an Agent

Within a Jadex agent there is only one global component namely, the reaction and deliberation mechanism which was briefly mentioned in the previous section. While it is possible to implement such mechanism manually, Jadex already provides a default deliberation strategy which allows intuitive specification and covers many popular application cases. The strategy is based on limit on the number of active goals of given type, also referred to as *cardinality* and *inhibition arcs* which define a partial order of importance between goals. The developer should take both of those into account while writing goal specifications since they provide a local perspective. The cardinality is only concerned with a single type of goal whereas inhibition arc expresses a local conflict. While defining arcs in our programs we have the option to specify them on the type level or on an instance level. The former would mean that as long as a goal of the first type is active no goal of the second type can be executed. Whereas the latter option requires the developer to specify an expression restricting to which specific goal instances the arc applies. We can also use arcs for establishing the order of processing between two goals of the same type.[3]

All the other components of the agent architecture are grouped into reusable modules called capabilities. This allows to group BDI elements which pertain to a specific functionality into a separate module. The implementation of agents can then be composed of those already existing modules. The enclosing capability of an element represents its scope, and given that an element only has access to elements of the same scope, flexible import/export mechanisms are needed to define external interface of the capability.[1]

The underlying agent platform places the incoming messages in the agent's global message queue. Before the message can be forwarded in the system, it needs to be assigned and handled by one or more capabilities. In the case where a message belongs to an ongoing conversation, an event for the incoming message will be created in the capability executing the conversation to which the message belongs to. A message can be identified as to whether it is a part of any already existing conversation by looking at its conversational id included as part of the message. Otherwise, if the message does not belong to any conversations, suitable capabilities have to be found. This is done by matching the message against event templates defined in the eventbase of each capability. These are then used to create appropriate events in the scope of the capabilities. Regardless of whether the message was new or already a part of conversation, the created events are subsequently added to the agent's global event list. The dispatcher will then select the appropriate plans for each of the events on that list. To do this, a list of applicable plans is generated by matching the event against the plan heads defined in the planbase of each capability. Only the capabilities where the event is defined should be considered. Next, a subset of the applicable plans is selected. The decision on which of those plans should be selected for execution is called meta-level reasoning. The default in Jadex is to only post messages to a single plan while for the case of goals many plans are executed sequentially unit the goal is completed. Any other internal events are posted to all plans at once. After the plans have been selected they are posted to a ready list and wait for execution.

The plans are then executed by a scheduler step-by-step. A step in that context is any sequence of basic actions which lead to an explicit waiting or affect the internal state of an

agent. Each step triggers an update in the agent's state which can lead to new internal or goal events.

Jadex adopts a hybrid approach when it comes to language as it explicitly distinguishes between the language used for static agent type specification and the language for defining the dynamic agent behaviour.[1] As a result agents in Jadex consist of two parts, firstly they are described by Agent Description Files (ADF) which denote the structures of beliefs and goals together with other implementation dependent details of the agent in XML syntax. Secondly, the activities they can perform are coded in plans which are ordinary Java classes.[5] Those classes can then access the BDI facilities of an agent through an API.[1]

Due to strong typing and explicit representation of elements in Jadex, the users are required to write detailed ADFs but in turn they are left with more rigorous consistency checking of agent models. Furthermore, to increase the expressive power of XML for things like object and queries creation, a separate expression language has been embedded, allowing to specify parts which are not easily expressed in XML. It has been extended with a subset of object query language and allows for creation of statements using the well-known *select-from-where* form.[1]

Let us now examine the syntactical aspects of some of the core concepts in the model. Beliefs in Jadex are represented in an object-oriented way allowing arbitrary Java objects to be stored as facts. As previously mentioned, these can be exported to make it accessible from an outer scope. A Java class for the facts of beliefs and belief sets must be defined. The user can also supply an initial fact data to configure the agent's mental state at creation time. The fact's value has to be stated in the expression language and can be declared as either static or dynamic, depending on the use case, for instance, dynamic facts are useful in the case where we need to represent values which are continuously being sensed from the environment. We can access beliefs and belief sets at runtime from within plans through operations on belief base or by issuing queries. The different goal types in Jadex can be summarized in an abstract base goal type. The concrete goal types can be later derived from this abstract implementation. Conditions like creation or drop can be defined as boolean expressions. It is also possible to define special parameters to transfer information between the goal's originator and processing plans. The declaration of plans themselves requires the specification of the plans heads for the circumstance under which the plan will be applicable. The pre and context conditions can be specified as Boolean expressions. Several parameterized plan type instances will sometimes be defined to ensure the goal achievement. The plan selection can also be influenced by setting priority value for it. The plan body requires that an expression is supplied to it in order to create a suitable plan instance. As of now, two different types of plan bodies are possible in Jadex: threaded and non-threaded. Both require an implementation of a Java class extending ThreadedPlan and providing the logic for the abstract *body()* method in which the domain-specific plan behaviour can be placed. Three other methods can be optionally implemented, namely *passed()*, *failure()* and *aborted()*. These would be called when plan processing has finished according to the plans final state.

Example Application

Disclaimer: Given that the example is simply meant to print out a “hello world” message, there was no real use in implementing plans or beliefs. Hence, the program below focuses more on how to get up and running with the platform and creation of a simple agent rather than demonstrating all of the concepts described in the previous sections.

Execution

The hello world program provided with this submission requires java 11 environment to be configured on the machine before execution. The project uses Maven therefore there is no need for any additional installations as the necessary dependencies are specified in the pom file. Simply open up the project in your IDE of choice and ensure that the maven dependencies are installed. Then proceed to run the program by executing the main method contained within the Hello class. The expected output is the “Hello world!” message being printed to the console.

Code

In order to start Jadex application we need to first start the Jadex Platform. This can be done by instantiating the PlatformConfiguration interface and using the PlatformConfigurationHandler to set up the configuration of the platform for us. Since this program is meant to be a simple demonstration I used the *getDefaultNoGui()* method for my configuration. I then proceeded by passing my component to the platform configuration object described above. An alternative approach could be to start the component using the platform’s ComponentManagementService however, once again for simplicity’s sake the former implementation was preferred. Our component in this case is the agent class HelloAgent.java which contains the agent implementation for the hello world program. After the platform is configured then platform is started using the *createPlatform()* method of the Starter class which takes our configuration as argument. The object returned by *Starter.createPlatform()* is called a *Future*. It represents a result that is not yet available - method calls returning a Future will typically return it instantly. Using the *get()* method will block until the result is available so we can work with it.

The Agent code is included in a separate class called HelloAgent.java. As per the documentation, since we are looking to implement a Micro Agents, i.e. the simplest type of component, we have to abide by certain requirements. Firstly, the name of the class has to end with “Agent” and secondly the class has to be annotated with the @Agent annotation. Optionally we can also include a @Description annotation which allows us to provide a description if our agent. Our agent implements one method called *executeBody()*, as the name suggests the method is intended to execute the functional body of the agent and is only called once. In our case the implementation includes a simple print statement after which we return an object of type *IFuture.DONE*. IFuture is a Jadex implementation of futures interface, which is similar to Java Future to an extent but also adds a listener notification mechanism. The *DONE* field is simply a future representing a completed action. The addition of the @OnStart annotation to this method ensures that it is called upon the start-up of the agent.

Conclusions

Overall I enjoyed the research process and learning about the Jadex framework. I was able to draw similarities to the languages covered throughout this module but I was also interested to see how they differed in certain aspects. I personally think Jadex is a very well designed framework. It's clear to me that the designers made the necessary considerations while thinking about its architecture and that it has been conceived with clear conceptual as well as technical goals in mind. It enhances the state-of-the-art BDI architecture by addressing some shortcomings of current BDI agent platforms such as implicit goal representation. However, in many cases developers would like to use a specific model which might fit their project requirements better, hence the BDI is not a once and for all solution and this also has been taken into account by Jadex. While this review has been focused only on the BDI model, Jadex does in fact support different architectures through the kernel concept. Furthermore, the use of established programming languages like Java and XML makes the programming of agents more accessible to inexperienced agent developers thus opening up the use of agent technologies to a wider audience. I'm not able to spot any obvious changes that I think would be needed for this framework in terms of its technical features or architecture as I believe I would need some more exposure and familiarity with agent systems programming to make a proper judgement. It appears as if at times the developer is expected to provide quite a detailed implementation logic like in the case of the ADFs which might prove challenging to inexperienced agent programmers like myself. However, as already mentioned in the case of ADF implementation it does pay off to do so as they are left with a more rigorous consistency checking mechanism of agent models than they would have been otherwise. Therefore, in conclusion, I do believe that Jadex is a very useful and high quality piece of software as not only does it offer a means for solving many problems in the multi agents system domain but also does so in a quite accessible way.

References

[1] Pokahr, Alexander, Lars Braubach, and Winfried Lamersdorf. "Jadex: A BDI reasoning engine." In *Multi-agent programming*, pp. 149-174. Springer, Boston, MA, 2005.

[2] Mascardi, V., Demergasso, D. and Ancona, D., 2005. Languages for Programming BDI-style Agents: an Overview. In *WOA* (Vol. 2005, pp. 9-15).

[3] Pokahr, A., Braubach, L. and Jander, K., 2013. The jadex project: Programming model. In *Multiagent Systems and Applications* (pp. 21-53). Springer, Berlin, Heidelberg.

[4] Braubach, L., Pokahr, A. and Lamersdorf, W., 2004, September. Jadex: A short overview. In *Main Conference Net. ObjectDays* (Vol. 2004, pp. 195-207).

[5] Eymann, T. *et al.* (2005) *Multiagent System Technologies: Third German Conference, MATES 2005, Koblenz, Germany, September 11-13, 2005, Proceedings*. Springer (Lecture Notes in Artificial Intelligence). Available at: https://books.google.ie/books?id=EzLzXV-drEgC.