Action library for JaKta

Giacomo Romagnoli
giacomo.romagnoli4@studio.unibo.it

August 2024

JaKta is an internal Domain-Specific Language (DSL) designed to support the definition of BDI agents in Kotlin [1]. Inspired by Jason [2], Jakta makes use of internal and external actions to practically extend AgentSpeak; however, since it is still a project in development, the user is responsible for independently implementing its own actions. The purpose of this project is to address the lack of predefined actions by adding a library, which the programmer can use as a starting point in the declaration of a Multi-Agent System (MAS).

1 Requirements

Since the JaKta project was inspired by the Jason language, an investigation was conducted during the analysis phase on the actions provided to the Jason programmer. This analysis involved identifying the set of internal actions present in the reference language, categorizing them, and finally assigning a priority to each one. The purpose of this research was to set up the implementation phase in an iterative manner, thereby defining the minimum requirements for the library. Here is a table that includes all the actions evaluated.

Action	Description
.desire(D, [I])	Checks whether D is a desire: D is a desire either if there is an event with +!D as triggering event or it is a goal in one of the agent's intentions I.
.drop_desire(D)	Removes desire D from the agent circumstance. This internal action simply removes all +!D entries (those for which .desire(D) would succeed) from both the set of events and the set of intentions. No event is produced as a consequence of dropping desires.

.drop_all_desires	Removes all desires of the agent. No event is produced. This action changes the agent's circumstance structure by removing from set of events all events for goals (like +!g -!g +?g) and then calling .drop_all_intentions. It does not remove external events (which are not considered as a desire)
.intend(G, [I])	Checks if goal G is intended: G is intended if there is a triggering event +!G in any plan within an intention I; just note that intentions can appear in E (list of events), PA (intentions with pending actions), and PI (intentions waiting for something) as well. This internal action backtracks all values for G.
.drop_intention(I)	Removes intentions to achieve goal I from the set of intentions of the agent (suspended intentions are also considered). No event is produced.
.drop_all_intentions	Removes all intentions from the agent's set of intentions (even suspended intentions are removed). No event is produced. This action changes the agent's circumstance structure by simply emptying the whole set of intentions (I), pending actions (PA), pending intentions (PI), and events in E that are not external events (thus generated by intentions).
.intention(ID, STATE, STACK, [current])	Returns a description of an intention. It is useful for plans that need to inspect some intention. The description of each item of the intention stack (i.e., an intended means) has the following form: im(plan label,trigger event,plan body term, unification function (a list of maps)).
.succeed_goal(G)	Removes goals G from the agent circumstance as if a plan for such goal had successfully finished. G is a goal if there is a triggering event +!G in any plan within any intention; also note that intentions can be suspended hence appearing in E, PA, or PI as well. The meta-event $\hat{!}$G[state(finished)] is produced.
.fail_goal(G)	Aborts goals G in the agent circumstance as if a plan for such goal had failed. An event -!G is generated. A literal G is a goal if there is a triggering event +!G in any plan within any intention; also note that intentions can be suspended hence appearing in sets E, PA, or PI of the agent's circumstance as well. The meta-event $\hat{!}$G[state(failed)] is produced.

.drop_event(D)	Removes events D from the agent circumstance. This internal action simply removes all +!D entries (those for which .desire(D) would succeed) from the set of events only; this action is complementary to .drop_desire and .drop_intention, in case a goal is to be removed only from the set of events and not from the set of intentions. No event is produced as a consequence of dropping desires from the set of events.
.drop_all_events	Removes all desires that the agent has not yet committed to. No event is produced.This action changes the agent's circumstance structure by simply emptying the whole set of events (E). This action is complementary to .drop_all_desires and .drop_all_intentions, in case all entries are to be removed from the set of events but not from the set of intentions.
.suspend(G)	Suspend goals G, i.e., all intentions trying to achieve G will stop running until the internal action.resume change the state of those intentions. A literal G is a goal if there is a triggering event +!G in any plan within any intention in I, E, PI, or PA. The meta-event $\hat{!}$G[state(suspended)] is produced.
.resume(G)	Resume goals G that were suspended by .suspend. The meta-event $\hat{!}$G[state(resumed)] is produced.
.suspended(G, R)	Checks whether goal G belongs to a suspended intention. R (a term) unifies with the reason for the suspend (waiting action to be performed, .wait, ...). The literal G represents a suspended goal if there is a triggering event +!G in any plan within any intention in PI or PA.
.abolish(Argument)	Removes all beliefs that match the argument. As for the - operator
.belief(Bel)	Gets beliefs in BB (excluding rules or inferred beliefs). Different from ordinary belief query that uses logical consequence to be verified, this internal action considers only the set of beliefs in the BB.
.findall(Term, Query, List)	Builds a List of all instantiations of Term which make Query a logical consequence of the agent's BB.
.setof(Term, Query, List)	Builds a Set of unique instantiations of Term which make Query a logical consequence of the agent's BB
.count(Pattern, Quantity)	Counts the number of occurrences of a particular belief (pattern) in the agent's belief base.
.namespace(N)	Checks whether the argument is a namespace.

.relevant_rules(Literal, Rules)	Gets all rules that can be used to prove some literal.
.list_rules	Prints out the rules in the belief base.
.add_plan(Plan, [Source], [Position])	Adds plan(s) to the agent's plan library.
.remove_plan(Label, [Source])	Removes plans from the agent's plan library.
.plan_label(P, L)	Unifies P with a plan term representing the plan labeled with the term L within the agent's plan library.
.relevant_plans(Trigger, Plans, [Labels])	Gets all relevant plans for some triggering event. This internal action is used
.relevant_plan(Trigger, Plan)	Gets relevant plans for some triggering event. This is a backtracking based version of .relevant_plans.
.list_plans([Trigger])	Prints out the plans in the plan library. List only plan that unifies the optional parameter as trigger event.
.send(Receiver, Ilf, Message, [Answer], [Timeout])	Sends a message to an agent.
.broadcast(Ilf, Message)	Broadcasts a message to all known agents.
.my_name(Name)	Gets the agent's unique identification in the multi-agent system. This identification is given by the runtime infrastructure of the system (local, saci, jade, ...).
.all_names(Names)	Get the names of all agents in the system. This identification is given by the runtime infrastructure of the system (local, saci, jade, ...).
.df_register(S , [T])	Register the agent in the Directory Facilitator as a provider of service S of type T (see FIPA specification). An optional second argument can be used to define the type of the service.
.df_deregister(S , [T])	Removes the agent in the Directory Facilitator as a provider of service S of type T (see FIPA specification). An optional second argument can be used to define the type of the service.
.df_search(S , [T] , L)	Unifies in L a list of all agents providing the service S of type T (see FIPA Directory Facilitator specification). An optional second argument can be used to define the type of the service.
.df_subscribe(S , [T])	subscribes the agent as interested in providers of service S of type T. For each new agent providing this service, the agent will receive a message ¡tell provider(Ag,Service)¿.

.member(T, L)	Checks if some term T is in a list L. If T is a free variable, this internal action backtracks all possible values for T.
.length(L, Length)	Gets the length of strings or lists.
.empty(L)	Checks whether the argument does not have any term.
.concat(Arg0, Arg1, ..., Result)	Concatenates strings or lists.
.delete(E, L, Result)	Delete elements of strings or lists.
.reverse(L, Result)	Reverses strings or lists.
.shuffle(List, Result)	Shuffle the elements of the List and unifies the Result.
.nth(N, L, E)	Gets the nth term of a list.
.max(L, M)	Gets the maximum value within a list of terms
.min(L, M)	Gets the minimum value of a list of terms
.sort(Ul, Ol)	Sorts a list of terms. The natural order for each type of terms is used. Between different types of terms
arity	functor
.list(L)	Checks whether the argument is a list
.suffix(S, L)	Checks if some list or string S is a suffix of list/string L. If S has free variables, this internal action backtracks all possible values for S.
.prefix(P, L)	Checks if some list/string P is a prefix of list/stringL. If P has free variables, this internal action backtracks all possible values for P.
.sublist(S, L)	Checks if some list S is a sublist of list L. If S has free variables, this internal action backtracks all possible values for S. This is based on .prefix and .suffix (try prefixes first then prefixes of each suffix).
.difference(S1, S2, S3)	S3 is the difference between the sets S1 and S2 (represented by lists). The result set is sorted.
.intersection(S1, S2, S3)	S3 is the intersection of the sets S1 and S2 (represented by lists). The result set is sorted.
.union(S1, S2, S3)	S3 is the union of the sets S1 and S2 (represented by lists). The result set is sorted.
.set.create(S)	Creates a java set represented by a variable.
.set.add(S, E)	Adds an element to set S.
.set.add_all(S, L)	Adds all element of list L to set S.
.set.remove(S, E)	Removes element E from set S.
.set.union(S, L)	Updates set S to be the union between itself and list L.
.set.difference(S, L)	Updates set S to be the difference between itself and list L.

.set.intersection(S, V)	Update set S to be the intersection between itself and V.
.set.clear(S)	Removes all elements from set S.
.queue.create(Q)	Creates a java Queue represented by a variable.
.queue.add(Q, E)	Adds an element to queue Q.
.queue.add_all(Q, L)	Adds all element of list L to queue Q.
.queue.head(Q, H)	H unifies with the head of queue Q.
.queue.remove(Q, E)	Removes all occurences of element E from queue Q.
.queue.clear(Q)	Removes all elements from queue Q.
.queue.create(Q, priority)	"Creates a priority queue represented by Q
.map.create(M)	Creates a java Map represented by a variable.
.map.put(M, K, V)	Puts a new entry in map M which has key K and value V.
.map.key(M, K)	Unifies K with all keys of M. If term K is ground, simply checks if K is a key in map M.
.map.value(M, V)	Unifies V with all values of M. If term V is ground, simply checks if exists a key associated with that value.
.map.get(M, K, V)	Unifies V with value mapped in M with key K.
.map.remove(M, K, V)	Removes mapping with key K from M, V unifies with the for K value.
.map.clear(M)	Removes all mappings from M.
.substring(Substring, String, [StartPosition], [EndPosition])	Checks if a string is sub-string of another string. The arguments can be other kinds of terms
.string(S)	Checks whether the argument is a string
.term2string(T, S)	Converts the term T into a string S and vice-versa.
.lower_case(S1, S2)	Converts the string S1 into lower case S2.
.upper_case(S1, S2)	Converts the string S1 into upper case S2.
.replace(S1, S2, S3, S4)	Replaces S2 by S3 in S1, result in S4.
.atom(A)	Checks whether the argument is an atom (a structure with arity 0)
.structure(S)	Checks whether the argument is a structure
.literal(L)	Checks whether the argument is a literal
.list(L)	Checks whether the argument is a list
.ground(G)	Checks whether the argument is ground, i.e., it has no free variables. Numbers, Strings, and Atoms are always ground.
.number(N)	Checks whether the argument is a number.
.type(X, T)	Retrieves types of the argument X.
.add_nested_source(Beliefs, Source, Result)	Adds a source annotation to a literal (used in communication).

.eval(Var, LogicalExpression)	Evaluates the logical expression (which computes to true or false), the result is unified with Var.
.at(When, Event)	Creates an event at some time in the future. This command is based on the unix "at" command
.wait(E, T)	Suspend the intention for the time specified by T (in milliseconds) or until some event E happens. The events follow the AgentSpeak syntax but are enclosed by { and }, e.g. {+bel(33)}, {+!go(X,Y)}.
.create_agent(Name, Source)	Creates another agent using the referred AgentSpeak source code.
.save_agent(FileName, [InitialGoals])	Stores the beliefs, rules, and plans of the agent into a file.
.kill_agent(Name, Deadline)	Kills the agent whose name is given as parameter. This is a provisional internal action to be used while more adequate mechanisms for creating and killing agents are being developed. In particular, note that an agent can kill any other agent, without any consideration on permissions.
.stopMAS	Aborts the execution of all agents in the multi-agent system (and any simulated environment too).
.fail	Fails the intention where it is run (an internal action that always returns false).
.perceive	Forces the agent architecture to do perception of the environment immediately. It is normally used when the number of reasoning cycles before perception takes place was changed (this is normally at every cycle).
.date(YY, MM, DD)	Gets the current date (year, month, and day of the month).
.time(HH, MM, SS, MS)	Gets the current time (hour, minute, seconds, and milliseconds).
.version(V)	Unifies V with the Jason version.
.range(Var, Start, End, Step)	Backtrack all values for Var starting at Start and finishing at End by increments of Step (default step value is 1).
.random(N)	Unifies N with a random number between 0 and 1.
.set_random_seed(N)	Sets the seed of Jason's random number generator.
.include(File)	Loads an .asl file, i.e., includes beliefs, goals, and plans from a file.
.printf(format, args...)	Used for printing messages to the console inspired by Java printf/format.

Following the analysis conducted on Jason, and in agreement with the JaKta developers, the following actions have been selected to be included in the library, as they are considered priorities.

- send/3
- broadcast/2
- my_name/1
- all_names/1
- random/1
- set_random_seed/1
- substring/3
- term2string/2
- lower_case/2
- upper_case/2
- replace/4
- atom/1
- structure/1
- list/1
- ground/1
- number/1
- type/2
- add_nested_source/3
- eval/2

1.1 Scenarios

The library is used by the user during the declaration of the MAS. In fact, the actions described above must be available by default to the programmer, who uses them in accordance with the JaKta syntax. It is therefore possible to include the execution of an action in a plan by specifying the name and parameters as follows:

```
mas {
    agent("sender") {
        goals {
            achieve("ask"("question"))
        }
        plans {
            +achieve("ask"("question")) then {
                execute("send"("receiver", "askOne", "good"(X)))
            }
        }
    }
    agent("receiver") {
        ...
    }
}
```

2 Design

2.1 JaKta's Actions

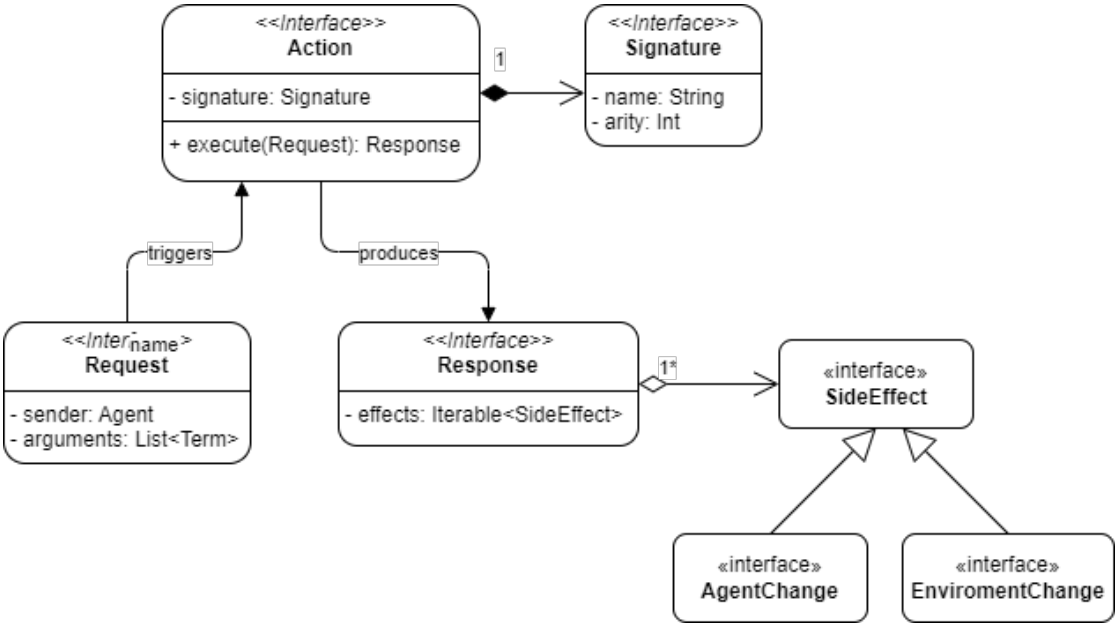

Figure 1: Action

As shown in Figure 1, actions in JaKta are triggered by a request and produce a response, which contains effects. Each effect describes a state change that must be applied, either to the agent if it is an AgentChange, or to the environment if it is an

EnvironmentChange. Each action is identified by a Signature, which defines its name and the number of parameters (logic terms). Unlike Jason, JaKta makes a clear distinction between internal and external actions, which differ based on the effects they produce. Internal actions only affect the agent that invokes them, while external actions impact only the environment. As shown in Figure 2, the actions were designed using a template

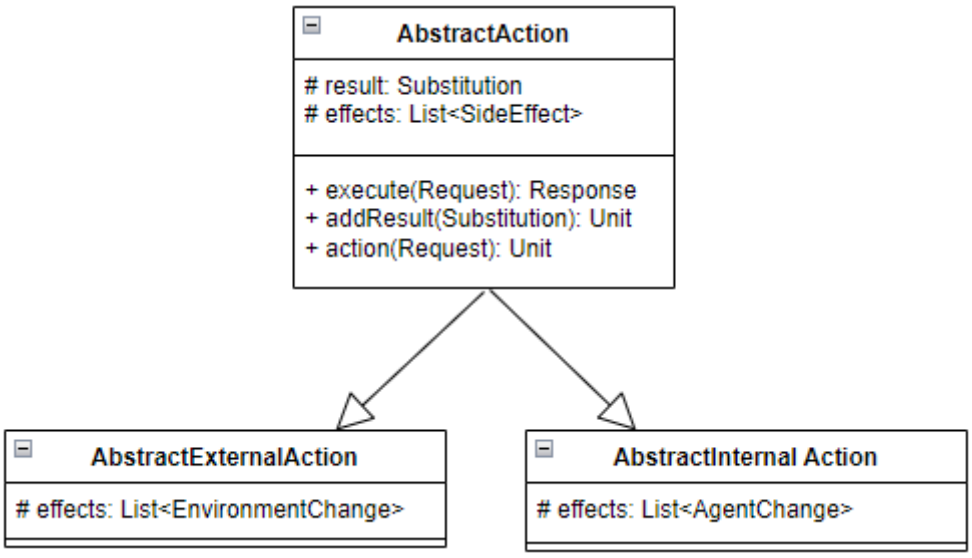

Figure 2: AbstractAction

pattern. In fact, there is an abstract class called AbstractAction, which is specialized into AbstractExternalAction and AbstractInternalAction. In the same UML diagram, you can see the field result, which represents the logical substitutions that must be applied after the action execution. The action method is the only abstract method (the template) implemented in the library actions to produce the desired effects, it is therefore necessary to define within the aforementioned method:

- reading and checking of input parameters
- one or more algorithms implemented using either the object-oriented or functional paradigm
- substitutions to be applied to logical variables
- other effects if necessary

2.2 The library

The library is implemented through two modules: InternalActions and ExternalActions, which respectively include internal and external actions, in the form of singletons. In order to include the entire library in the MAS, each module exposes a default method

that returns a collection of the actions it contains. The inclusion takes place during the MAS build phase (as shown in Figure 3 with a partial view), specifically in the build method of AgentScope and EnvironmentScope, since internal actions belong to individual agents, while external actions are part of the environment. In the following sections, the actions identified as necessary and listed in section 1 are further explored in detail.

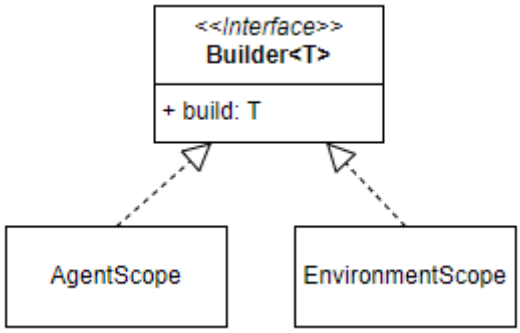

Figure 3: Builder

2.2.1 Type checking actions

This category includes all actions that aim to provide information or perform checks on the type of arguments, and therefore are internal actions. Specifically, they include:

- atom/1
- structure/1
- list/1
- ground/1
- number/1
- type/2

For all these actions, input validation constitutes the core algorithm, implemented using the 2pkt API, which is the Prolog implementation JaKta relies on to provide logical paradigm functionality and data types. From the perspective of effects, type-checking actions do not produce changes on the agent, but they can cause the failure of the executing plan. In fact, the first five actions do not apply substitutions unless the predicate (self-explanatory) is false. In such a case, the action ends with a failed substitution, resulting in the failure of the current plan. On the other hand, type/2 is responsible for unifying the second argument with an atom that describes the type of the first argument. Therefore, its result consists of a substitution generated by the unification between the

type of the first argument and the second argument. If the two terms cannot be unified, the result is a failed substitution.

2.2.2 Random generation actions

This group of actions includes:

- random/1
- set_random_seed/1

Unlike type-checking actions, these two internal actions are not strictly predicates, which is why input validation is crucial to understanding their behavior. Random/1 is responsible for generating random numbers between 0 and 1, and thus its single argument must be an unbound variable. In this case, the action returns a substitution for that variable with the randomly generated number. In all other cases, the action returns an empty result. The random number, whose generation represents the algorithm, is obtained using Kotlin's standard library. The execution of set_random_seed/1 results in a side effect on all future calls to random/1, setting a seed for random generation, as the name suggests. To achieve this behavior, both actions were implemented as full-fledged classes rather than singletons, allowing one to store a reference to the other and modify its state. The default method of InternalActions adds an instance of both actions to the collection, ensuring that the constructor of set_random_seed/1 receives the instance of random/1. It is clear that set_random_seed/1 does not produce any result (it does not apply substitutions) and, like random/1, has no other effects on the agent. Additionally, its argument must necessarily be an integer.

2.2.3 String manipulation actions

Traditionally in Jason, the string type is modeled as a first-class entity, distinct from logical terms. In JaKta, however, the string type and the logical term atom are equivalent, meaning there are no true strings, only atoms. For this reason, the actions:

- substring/3
- term2string/2
- lower_case/2
- upper_case/2
- replace/4

that manipulate strings in Jason actually operate on atoms. Substring/3 checks that the first argument, "Substring," is a substring of the second argument, "String." Additionally, it unifies the third argument, if present, with an integer representing the position in "String" where "Substring" begins. Interestingly, the validation of the arguments for this action is not as strict as one might expect. In fact, "String" and "Substring" are

evaluated as Kotlin strings by the algorithm, which handles checking for the occurrence of one within the other and determining the position. The result is a failed substitution if the predicate is false, a null result if the third argument is absent, or a substitution produced by unifying the third argument if present. Term2String/2, originally used to convert terms to strings and vice versa, takes on a different meaning in JaKta, translating to the conversion between any term and an atom. The algorithm is thus always a unification, but since term2string/2 is a true predicate, this process can work in both directions. In other words, if the first argument, "Term," is ground, the second argument, "String," unifies with "Term" converted to an atom (a process that is always possible for any term), producing the action's result. Conversely, if "Term" is not ground and "String" is an atom, the result is produced by unifying "Term" with the atom parsed as a term. Any other case results in a failed substitution. Lower_case/2 and Upper_case/2 are essentially two ways of calling the same action, as they represent the same predicate interpreted in opposite directions. Lower_case/2 relates the two arguments, "String1" and "String2," such that "String2" is the lowercase representation of "String1," while Upper_case/2 relates "String1" and "String2" such that "String2" is the uppercase representation of "String1." This means that the following holds: lower_case(X, "string") is equivalent to upper_case("string", X). To maintain consistency with the original design of these actions (not interpreted as predicates in Jason), both are available in the library but extend from a common class, differentiated only by a flag that indicates the direction of the transformation. The algorithm always involves either a lowercase-to-uppercase or uppercase-to-lowercase conversion, and the result is produced by unifying one of the arguments with the transformation of the other. Therefore, at least one of the arguments must be an atom; otherwise, a failed substitution is returned. Replace/4 is responsible for replacing "S2" with "S3" in "S1" and unifying the result of the transformation with "S4," where "S1," "S2," "S3," and "S4" are the arguments in order. The input validation is strict, as the first three arguments must be atoms for the algorithm—replacing all occurrences of a substring with another—to be applied. The result is always determined by applying substitutions to "S4" to unify it with the modified "S1." If this is not possible, a failed substitution occurs. All string manipulation actions are internal and do not produce any effects on the agent beyond their result. All algorithms were developed using Kotlin's standard library.

2.2.4 Communication utility actions

This group of actions includes both internal and external actions, useful for communication between agents but not directly, meaning without exchanging messages. The actions in question are:

- my_name/1
- all_names/1
- add_nested_source/3

My_name/1 is an internal action useful for an agent to retrieve its own name. Simply put, the single argument unifies with the agent's name, producing the result of the execution. To provide this functionality, the request triggering this action must be utilized. This request provides, in addition to the arguments, a reference to the agent, which allows the agent's name to be extracted. Once again, through the triggering request, it is possible to identify the name of the requesting agent and, similarly, to retrieve the collection of all other agent names. Requests are specialized as either internal or external, each providing different information. In the case of external requests, a reference to the environment is available, from which the names of all agents can be retrieved. The algorithm simply unifies the list of names obtained from the environment, excluding that of the requesting agent, with the single argument, thus producing the action's result. Add_nested_source/3 is an internal action used to manipulate the source of a belief, either for the agent's internal purposes or to use the resulting belief as a message. The source is a term in the form source(sender) that specifies the origin of a particular belief. In JaKta, this term is conventionally added to every belief as the first argument to indicate its source. Add_nested_source/3 is responsible for nesting the source "Source" (the second argument) within the belief "Belief" (the first argument), or alternatively adding it if "Belief" does not already have one. To perform this task, "Belief" must be a structure, and "Source" must be an atom. The algorithm creates a new term by modifying "Belief." If it does not have a source, it adds source(Source) as the first argument. Otherwise, it nests the existing source in the form source(sender, source(Source)). The third argument is then unified with the modified belief, generating the result. If "Belief" is a list, the algorithm applies to each element in the list, producing a new list that is unified with the third argument, just as in the simpler case, to produce the result. None of the actions in this category produce any effects on the agent.

2.2.5 Messaging actions

This category includes the two actions

- send/3
- broadcast/2

that constitute the only tools available for exchanging messages between agents. It should be noted that both actions take the illocutionary force (or ILF) as an argument, which strongly changes the semantics of the message for the recipient(s). Due to the complexity of the implementation, not all Jason ILFs are handled, but the most useful ones have been selected to enhance the usability of JaKta, namely: tell, achieve, and askOne (the latter available only for send/3). The goal of broadcast/2 is suggested by its name: to forward a message, represented by a logical term, to all other agents in the MAS. The effect of receiving such a message is determined by the ILF, an atom that can take the values tell and achieve. In the case of a broadcast tell, each recipient adds the message to their belief base, while in a broadcast achieve, each recipient adds the message to their goals. To achieve these behaviors, an EnvironmentChange is used,

which, in both forms of broadcast/2, is an external SideEffect of type BroadcastMessage. Once the action is resolved, the JaKta engine translates the effect into behavior. There is no need to produce results, even in cases where unsupported ILFs are provided, in which case the action produces no effects. In send/3, an additional argument appears: the name of the recipient. Regarding the ILFs tell and achieve, the execution is similar to broadcast/2, with the difference that the EnvironmentChange generated is of type Message and is forwarded only to the specified agent by name. However, the behavior of send/3 changes drastically when used in the askOne format. In this case, the "Message" argument takes on a completely different meaning—it becomes a question, expressed as an incomplete logical term. The goal of send/3 askOne is to forward the question to the named agent and then suspend the current intention while waiting for a response. The expected response is the same question, made ground by the recipient. To achieve this behavior, two SideEffect are produced: the first is always of type Message, ensuring the message and ILF are delivered to the recipient, and the second is of type SuspendUntil. The latter is accompanied by the event that triggers the resumption of the intention, suspending it until the response is received. The response is then unified with the "Message" argument. Given the synchronous nature of send/3, it is essential for the proper functioning of the system that a response is received from the recipient. The responding agent must provide a plan whose header unifies with the question and fills in the free variables appropriately, aligning with the purpose of the communication. send/3 does not produce results and does not handle unspecified ILFs. To enable synchronous agent interaction via send/3, several changes were made to the core of JaKta, which are covered in the JaKta's core changes section of this report.

2.2.6 Other actions

In this category there is only

- eval/2

as it doesn't fit into any of the previous categories. Eval/2 is responsible for evaluating the logical expression in the second argument, which is represented as a logical structure. The algorithm checks the validity of the expression against the belief base of the agent making the request. Similar to previously discussed actions, the belief base is accessed through the reference to the agent contained in the request. Once the expression is evaluated, the first argument is unified with the result (either true or false), thus determining the substitution to be applied. This action does not produce any effects on the agent.

2.3 JaKta's core changes

As previously mentioned, several modifications have been made to JaKta's engine, where "engine" refers to the entirety of the JaKta system, excluding the DSL and the action library. First, the SuspendUntil side effect was introduced. The particularity of this effect, which is produced by messaging actions, is its dual role as both an internal

and external effect. This property is achieved through multiple inheritance, where SuspendUntil implements both the AgentChange and EnvironmentChange interfaces. This allows send/3, an external action, to modify the agent's state, breaking JaKta's standard logic of action responsibilities. SuspendUntil is simply a plain object containing an Event representing the receipt of a response, which is useful in each agent's sense-deliberate-act cycle (implemented in the AgentLifeCycle class) to suspend the current action, setting its resume condition. Subsequent modifications were made to JaKta's intentions, modeled as first-class entities. Fields were added to the Intention state to model a suspended intention awaiting a specific event. Accordingly, the routines that make up the sense and act phases were modified, ensuring that during the agent's sense phase, suspended intentions are unlocked if the event they are waiting for is perceived, and that during the act phase, suspended intentions are not executed. The deliberate phase was also changed to allow the programmer to avoid specifying the send of a response in a synchronous communication. This change consists of adding a subroutine to deliberate, which activates when a test event (generated by receiving a message with ILF askOne) is associated with a plan and an intention, adding the send of a response (where the message is always identical to the plan's header) as the last instruction of the plan. Additionally, a bug was fixed that caused an exception to be thrown when the last instruction of the last plan of an intention was a test on the belief base. To ensure the highest code quality, further refactoring was carried out on the aforementioned sections of JaKta.

3 Self-assessment / Validation

To demonstrate the achieved objectives and their compliance with the specifications, automated tests were produced for actions that do not have SideEffect. For more complex actions, use cases were developed to exemplify their usage. In cases where automated tests were not available for certain actions, manual testing was conducted.

4 Conclusions

This project aims to enhance the JaKta environment by adding a library of actions useful to the programmer. In addition to a handful of essential actions, it also provides substantial language features, such as synchronous communication.

4.1 Future Works

In the future, the library could be expanded and reorganized to meet the ambitions of the JaKta project. Additionally, it will be possible to enhance the language engine by adding new features or modifying existing ones.

4.2 What did we learned

During the development, I had the opportunity to delve into the BDI architecture by studying a possible implementation, which allowed me to gain a pragmatic understanding of the architecture itself, as well as agent-based programming. I discovered that it is an interesting and powerful paradigm, though still not fully matured for mainstream adoption.

References

[1] Martina Baiardi, Samuele Burattini, Giovanni Ciatto, and Danilo Pianini. Blending bdi agents with object-oriented and functional programming with jakta. *Springer Nature Computer Science*, 2024.

[2] Jason, a java-based interpreter for an extended version of agentspeak. `https://jason-lang.github.io/`.