

Kotlin-BDI

Marco Galassi

marco.galassi6@studio.unibo.it

March 2021

Abstract

The project aims to create a Kotlin library which can be used to develop systems based on the interaction of independent and lightweight agents placed inside a common environment. The context of the project is the research in the fields of Autonomous Systems and Multi-Agent Systems (MAS). The definition of agent is substantially based on the one presented by Anand S. Rao in his famous design for BDI (Belief-Desire-Intention) agents for the AgentSpeak(L) language. The agents' reasoning procedure and perception of the environment are modelled through logical structures, as defined in Prolog.

1 Goal and requirements

The *Kotlin-BDI* project's aim is to provide an useful tool for researchers and developers to ease the creation of complex systems where autonomous BDI agents can interact inside a shared environment. The notion of agent is taken directly from the definition specified in the AgentSpeak(L) (from now on AgentSpeak) language by its creator Anand S. Rao. For this reason *Kotlin-BDI*'s real goal is to be an actual implementation of AgentSpeak in Kotlin language, by following - and enhancing - the original design.

From a design point of view, *Kotlin-BDI* should mimic in an elegant way every basic AgentSpeak structure and concept and eventually let developers write BDI agents *naturally*, as if they were working with a pure logic programming language. For this reason, the library must provide a tool to let anyone use it easily and flexibly and it should also hide any kind of complexity away from the user.

From a technical point of view, *Kotlin-BDI* should always follow the Agent-Oriented programming paradigm and thus its agents should always be considered autonomous and independent entities. The project aims to make them as lightweight as possible, too. The library should also provide basic mechanisms for agents' interaction and perception in the form of an underlying environment. The main requirements can be specified as such:

- The library must provide a complete and functional implementation of (at least) the basic AgentSpeak language features in Kotlin.
- The library must provide a sort of DSL (domain-specific language) to ease the way a developer can interact with it. The DSL should be modelled to resemble logic programming and should enforce a natural way to think about agents.
- Agents should follow the original AgentSpeak design and must always be conceived, from the very start, as autonomous and concurrent entities.
- The library should provide the tools to create environments with a vast number of agents, which should be made as lightweight as possible.
- The library should provide some design of the notion of *environment* upon which agents live and where agents produce and perceive a concept of *change*.

1.1 AgentSpeak

The AgentSpeak abstract language has been a fundamental milestone in the Autonomous Systems and Reasoning Agents fields of study. Its purpose was to define a way to easily describe and program BDI agents capable of high-level reasoning, goal planning and event reacting. The language is proposed by Rao in *AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language*[1] as an *abstraction* of already existing BDI systems, aiming to fill the *large gap between theory and practice*.

The term *BDI* (meaning *belief, desire and intention*) had previously been made popular roughly ten years prior to Rao's paper by the works of Michael P. Georgeff and Amy L. Lansky on Procedural Reasoning Systems (PRS), whose main goal was to address the problem of reasoning in dynamic environments, where new constraints may continuously arise. There was the need to define intelligent agents which could *react* to the external stimuli (given by the environment itself or other agents) and build a *procedural knowledge about how to function in the world*. In *Procedural Knowledge*[2] and especially in *Reactive Reasoning and Planning*[3] the authors explain that *the kind of behaviour expected of a rational agent* can be expressed as the result of the *manipulation and reasoning* about the *psychological attitudes of belief, desire, and intention*, thus the name. To sum it up with a direct citation from [3]:

The system, called a Procedural Reasoning System (PRS), is endowed with the attitudes of belief, desire, and intention. At any given instant, the actions being considered by PRS depend not only on its current desires or goals, but also on its beliefs and previously formed intentions. PRS also has the ability to reason about its own internal state - that is, to reflect upon its own beliefs, desires, and intentions, modifying these as it chooses.

Heavily inspired by PRS and dMARS (Distributed Multi-Agent Reasoning System), AgentSpeak's main purpose is to define a *reasonably useful and expressive language* to *make the task of agent programming easier*. The language consists of certain building blocks:

- *beliefs* (or *belief base*), representing the internal knowledge the agent owns regarding itself, other agents and the surrounding environment.
- *goals*, which are *states of the system which the agent wants to bring about* representing the *adopted desires* of an agent. These can be divided into *achievement* goals (ie. what the agent wants to do) and *test* goals (ie. what the agent wants to know).
- *events*, which represent the way the agents perceive a change in the environment (*external event*) or the acquisition of a new goal (*internal event*).
- *plans*, specifying *the means by which an agent should satisfy an end*. A plan can be seen as a sequence of goals and actions (here called *formulae*) the agent tries to complete or perform after the perception of an event. The actual adoption of one or more plans as the intended means to fulfill a goal is called *intention*.
- *actions*, which represent the way the agents can affect the surrounding environment, possibly changing its state.

The AgentSpeak operational semantics revolves around the concept of the agent *configuration*, which is mainly the combination of beliefs, events, plans, actions and intentions any agent knows and possesses at any given time. This is formally defined through the $\langle E, B, P, I, A, S_E, S_O, S_I \rangle$ tuple, where E is the set of events, B is the set of beliefs, P is the set of plans, I is the set of intentions, A is the set of actions and S_E, S_O and S_I are selection functions operating on the set of events, on the set of applicable plans (for the selected event) and on the set of intentions, respectively.

Based on its state during evaluation, the agent then follows one step of a precise procedure which, on the whole, lets the agent process incoming events and considering what it knows about itself and the environment decide which plan is the most appropriate to react with. The agent then follows the goals and executes the actions specified in the selected plan, creating (or updating) its intentions. The practical ways to do all these steps are described in the paper (with the help of the following pseudocode provided by Rao himself).

```

Algorithm Interpreter()
while  $E \neq \emptyset$  do
   $\epsilon = \langle d, i \rangle = \mathcal{S}_E(E)$ ;
   $E = E/\epsilon$ ;
   $O_\epsilon = \{p\theta \mid \theta \text{ is an applicable unifier for event } \epsilon \text{ and plan } p\}$ 
  if external-event( $\epsilon$ ) then  $I = I \cup [\mathcal{S}_O(O_\epsilon)]$ ;
  else push( $\mathcal{S}_O(O_\epsilon)\sigma, i$ ), where  $\sigma$  is an applicable unifier for  $\epsilon$ ;
  case  $first(body(top(\mathcal{S}_I(I)))) = true$ 
     $x = pop(\mathcal{S}_I(I))$ ;
    push( $head(top(\mathcal{S}_I(I)))\theta \leftarrow rest(body(top(\mathcal{S}_I(I))))\theta, \mathcal{S}_I(I)$ ),
    where  $\theta$  is an mgu such that  $x\theta = head(top(\mathcal{S}_I(I)))\theta$ ;
  case  $first(body(top(\mathcal{S}_I(I)))) = !g(t)$ 
     $E = E \cup \langle +!g(t), \mathcal{S}_I(I) \rangle$ 
  case  $first(body(top(\mathcal{S}_I(I)))) = ?g(t)$ 
    pop( $\mathcal{S}_I(I)$ );
    push( $head(top(\mathcal{S}_I(I)))\theta \leftarrow rest(body(top(\mathcal{S}_I(I))))\theta, \mathcal{S}_I(I)$ ),
    where  $\theta$  is the correct answer substitution
  case  $first(body(top(\mathcal{S}_I(I)))) = a(t)$ 
    pop( $\mathcal{S}_I(I)$ );
    push( $head(top(\mathcal{S}_I(I))) \leftarrow rest(body(top(\mathcal{S}_I(I))))$ ,  $\mathcal{S}_I(I)$ );
     $A = A \cup \{a(t)\}$ ;
endwhile.

```

Figure 1: AgentSpeak interpreter pseudocode by Rao

2 Analysis

The main requirement of *Kotlin-BDI* is to provide an implementation of the AgentSpeak abstract language in Kotlin, which of course can only be done after an extensive and careful study of the original paper by Anand S. Rao.

Upon reading, it is self-evident that BDI agents must be defined as entities which represent their knowledge (here called *belief base*) as logical structures, perceive anything that happens as something which can be reduced to true or false values and plan their actions as sequences of atomic deductions on what they already know or they need to discover. Thus it is immediately clear that any implementation of AgentSpeak agents should rely, at least in its core, on a programming language which follows the logic programming paradigm. Kotlin, despite being a general purpose language, does not support logic programming in itself and this requires a tool (as a library or a piece of software) to fill the gap.

The paper is straightforward in explaining the definition of the agents' structure and behaviour but it often avoids to give specific details regarding actual implementations. Most importantly:

- The concept of *environment*, which can be seen as the container in which all the agents live and, at the same time, the host of anything that can be perceived and capable of change and again it can also be considered the entity which enables communication between the agents;
- The concept of *action*, which is the actual way an agent can produce any kind of change in the environment.

The fact that agents must be designed as autonomous entities heavily implies that any kind of communication between them or the environment must be based on asynchronous message passing. Any entity that is open to communication must then provide a way to receive messages, typically through a channel. Although it is not specified in the original paper, in *Kotlin-BDI* communication should always be *one-way*: one agent should never wait for a reply from the other entity it sent the message to, nor it should expect a response at all. Any kind of interaction must always be designed with this concept in mind.

3 Design

3.1 Structure

Kotlin-BDI lets users build systems based on the interaction of two main entities: environments and agents. The interaction should not be strictly required, though. Being designed as totally independent entities, agents and environments do not depend on one another, although complex features may require some kind of orchestration between them.

Structurally speaking, environments can be seen as special *containers* for at least two reasons:

- They can be seen as the stage on which multiple agents act, a sort of permanent surrounding. An environment can thus contain one or more agents.
- They keep track of everything that happens outside the private scope of the agents, as a sort of common and shared state which can be perceived. This concept of state must be defined and managed inside the environment in a data structure of some kind.

Agents are the proactive entities of the system and their structure revolves around the concepts of *state* and *execution*.

The state (or *configuration*) is the data structure upon which the agent bases its reasoning on. In *Kotlin-BDI* this is basically the evolution of the tuple seen in the original paper and can be described, at a given moment, as the composition of certain beliefs, events, plans and intentions. It also contains the reference of the currently selected event, plan or intentions and the set of relevant and applicable plans as well.

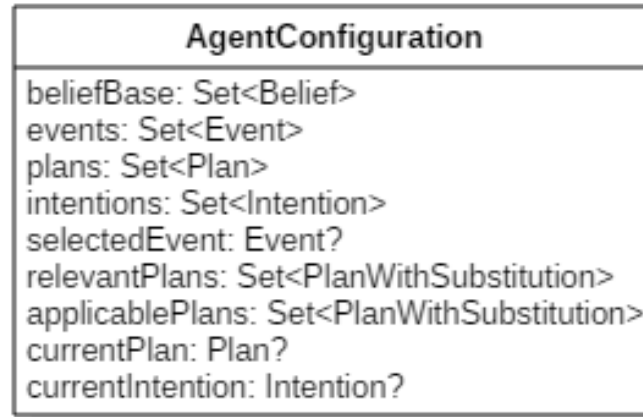


Figure 2: AgentSpeak interpreter pseudocode by Rao

To achieve autonomous *execution* and asynchronous communication *Kotlin-BDI* must provide to any agent a private flow of control and a non-blocking channel, respectively. The library aims to let users create agents in a flexible way and for this reason certain execution (what does it mean for an agent to execute?) and termination (what does it mean for an agent to stop?) policies can be statically defined.

Agents capabilities can be enhanced by defining particular *actions*, which can then be called in a plan body. These actions can be internal (which can be fulfilled in an agent private scope) or external (which are conveniently placed in the environment scope and can be used by any agent). The result of an action can be coded in Kotlin in *plugins* (for internal actions) and *artifacts* (for external ones), which can be linked to agents and environments, respectively.

3.2 Behaviour

The overall behaviour of *Kotlin-BDI* systems mainly depends on the behaviour of its agents, which is primarily defined by their *state* and their *reasoning procedure*.

The reasoning procedure describes how an agent should act based on its current state at the moment of evaluation. It is called a “procedure” because it aims to strictly define what should happen on certain state configurations, sort of a sequence of steps (formally called *execution steps*) that must be followed. *Kotlin-BDI* lets users customise the agents behaviour by specifying any kind of reasoning procedure and it also provides, by default, an implementation of the original AgentSpeak interpreter.

Any reasoning procedure implementation must define, by contract, a way to execute one of its step singularly and another way to execute the whole procedure entirely. This is an important detail: by design, *Kotlin-BDI* agents may be run at different paces and developers may want them to execute one step at a time, maybe in a controlled fashion. Thus, the concept of step must be defined by the user and must follow a single requirement: it must work on an instance of the agent state and return the new resulting state. By continuously executing steps the agent can perform any kind of reasoning which always result in a change of its inner state.

3.3 Interaction

Agents and their surrounding environment communicate asynchronously by exchanging messages. These messages are modelled as a logic data structure as well and specify in logical terms the message sender, receive, action and payload. The action is what conveys the semantic of the message (what it represents), while the payload contains the actual information.

Based on a message action the environment can react in the following ways:

- If the message is a *whisper*, the environment carries it to the specified receiver (1-to-1 communication).
- If the message is a *shout*, the environment carries it to every other agents it knows (1-to-N communication).
- If the agent asked the environment to *perform*, the latter tries to find a suitable artifact to execute the requested action.

By whispering and shouting, agents can actively communicate meaningful messages and can send each other *triggering events* to inform that something changed (so that they can update their beliefs) or to ask to do something (by trying to pursue some kind of goal).

The environment can send these events too. This should happen whenever something change in its state - allowing the agents to *perceive* that and act accordingly - or as a result of some artifact execution.

4 Implementation Details

4.1 Technologies

The project is entirely written in Kotlin and uses Gradle as its compile and build tool. Being a general-purpose language, Kotlin does not support logic programming by default.

The bridge between Kotlin and Prolog (the project language of choice for the AgentSpeak implementation) has been made possible thanks to the 2pKt[4] library and the tuProlog[5] library at its root, both made by the UniBo APICe group. The contribution given by these libraries to *Kotlin-BDI* has been fundamental because they provide a layer of Kotlin-defined structures on top of a fully functional Prolog engine. Logic programming concepts as such as **Term**, **Struct**, **Substitution** or **Unifier** are wrapped in Kotlin classes, making the integration really straightforward.

The *Kotlin-BDI* project has been originally developed as a fork of the Coordination with TuSoW[6] project, whose main purpose is to enable coordination and communication between different processes or MAS through tuple spaces written in Linda. *Kotlin-BDI* bases its concept of agents as “executable lightweight concurrent processes” on several classes (which will be described in the next subsection) defined there.

4.2 Agent

4.2.1 Execution and state update

Let’s now discuss *when* an agent actually executes. *Kotlin-BDI* agents’ control flow can be customised by declaring a certain *dispatcher* to be used for them. The dispatcher is a concept built on top of the agency and control modules provided by the Coordination with TuSoW project and relies on one of the different implementations of the **Runner** interface. By using a specific runner, the user can make an agent run asynchronously or synchronously (i.e. sharing its control flow with the environment), in a thread or in a Kotlin coroutine, periodically or continuously, and so on. Modelling the agent as an **Activity** and coding what should happen at every “tick” of the runner should be the only two requirements.

Technically speaking, the dispatcher will end up calling some specific methods of the underlying **Controller** inside the **Activity**. These methods are:

- **onBegin**, called once at initialisation.
- **onStep**, called at every “tick” of the dispatcher’s **Runner**.
- **onEnd**, called once at termination.

The following code snippet shows how *Kotlin-BDI* agents perform any kind of reasoning:

```

1  // Consider pending events, which must be merged with the existing ones
2  val currentState = managePendingEvents(currentState())
3  // Perform the procedure
4  val newAgentState = when(executionPolicy) {
5      ExecutionPolicy.OneStepAtATime ->
6          reasoningProcedure.performSingleReasoningStep(currentState)
7      ExecutionPolicy.WholeProcedureAtOnce ->
8          reasoningProcedure.performProcedureEntirely(currentState)
9  }
10 // Update the state
11 updateState(newAgentState)
12 // Decide whether to stop or to continue
13 when(terminationPolicy) {
14     TerminationPolicy.StopAfterOneExecution ->
15         controller.stop(newAgentState)
16     TerminationPolicy.ContinueIndefinitely ->
17         controller.`continue`(newAgentState)
18 }

```

This piece of code is placed inside the `onStep` method and thus it will be executed whenever the underlying dispatcher calls it. In order:

1. Any pending incoming events are managed and eventually added to the events set.
2. Based on the defined execution policy, the agents performs one single execution step of its reasoning procedure or executes it entirely at once.
3. The agent updates its inner state with the one resulted from the previous execution.
4. Based on the defined termination policy, the agents stops or continues (by calling the appropriate method on the underlying controller).

4.2.2 Reasoning procedure

The reasoning itself is completely delegated to the agent's reasoning procedure. *Kotlin-BDI*, by default, provides a FSM (finite-state machine) implementation of the reasoning procedure, reflecting the original AgentSpeak pseudocode. This is composed of 11 execution steps, almost always modelled as extension functions of the `AgentConfiguration` interface and returning another `AgentConfiguration` instance. The sequence is as follows:

1. **Idle**: if the events set is not empty, go to step 2. If it is and if the intentions set is not empty, go to step 9. If it is, go to step 1.

2. **Select event:** select one event from the events set using the provided selection function.
3. **Update belief base:** update the agent belief base according to the selected event.
4. **Find relevant plans:** considering the selected event, find all the relevant plans in the plans list.
5. **Find applicable plans:** considering the relevant plans, find all the applicable plans among them.
6. **Select applicable plan:** select one applicable plan from the applicable plans set using the provided selection function.
7. **Process the selected event:** considering the selected event and selected applicable plan, a new intention is generated and added to the intentions set.
8. **Consider the intentions set:** if the intentions set is not empty, go to step 9. If it is, go to step 1.
9. **Select intention:** select one intention to execute from the intentions set using the provided selection function.
10. **Execute intention:** execute the selected intention by processing the first formula of the body of its top plan.
11. **Cleanup:** clean the agent configuration by removing the completed intentions from the intentions set. Go to step 1.

4.2.3 Plan body formulae and actions

Being an implementation of AgentSpeak, *Kotlin-BDI* users can specify different kinds of formulae in an agent's plans body. Some of these are built by following the instructions presented in the original paper, namely:

- Achievement goal (definition 13).
- Test goal (definition 14).
- Action (definition 15).
- *true* (definition 16).

The library adds another two:

- Belief addition/deletion.
- Parallel achievement/test goal.

As it can be read in *An Operational Semantics for a BDI Agent-Oriented Programming Language*[7], the addition and deletion of beliefs were pointed out by Rao (but eventually excluded from the list of possible formulae). The parallel execution of a goal just requests the agent to create a brand new intention with the provided goal, effectively as if the goal was a result of an incoming external triggering event.

The concept of action is not explicitly defined in Rao’s paper. Yet what they should do is clear: they *may change the state of the environment*. That implies that these actions may produce some effects outside the scope of the executing agent. These are called *external* actions. *Kotlin-BDI* supports another kind of actions, completely encapsulated in the agent scope: the *internal* ones.

Actions, by design, enhance agents and environments capabilities and the library provides a convenient and flexible way to express them. Developers can write anonymous classes in Kotlin with methods that can be called at run-time through reflection. These classes must always implement the generic **Feature** interface and provide a unique identifier to be used by agents and environments. A **Feature**’s method can eventually be invoked if an action mapping its signature is executed. *Kotlin-BDI* defines two different **Feature** implementations:

- **Plugin**, which can be collected in a **PluginLibrary** and linked to an agent to be used during an *internal* action. **Plugins** methods must define an **Agent** instance as the first parameter. **Plugins** methods may also return a **Substitution**, which can then be used to continue the process of unification along the remaining formulae in a plan body.
- **Artifact**, which can be collected in a **Workspace**[8] and linked to an environment to be used when one of its agents tries to execute an *external* action (by sending a *perform* message). **Artifacts** methods must define an **Environment** instance as the first parameter and an **Agent** instance as the second one. **Artifacts** methods may also return an **EnvironmentData**. If that is the case, the environment can update its inner state with it, detect any changes and propagate them to its agents as triggering event. The original purpose for external actions is thus preserved.

Being written in Kotlin, a **Feature**’s method can also throw an exception during its execution. Under normal circumstances, a thrown exception is considered an irremediable error and thus it interrupts the calling agent’s flow of control. But there is always the need for developers to throw an exception in a “graceful” and manageable way which does not stop the agents from continuing their work. For this reason, users may throw the special **ActionException** to prevent any complete halt. Whenever that happens, the agents just considers that action as a failed formula and goes on.

4.3 Environment

The environment holds its inner state as an instance of **EnvironmentData**, which is implemented as a simple map pairing **Strings** with **Any** kind of value. By design, there is no enforcing about the value types because their handling must be managed by developers.

The state is modelled as an **Observable** variable. This is extremely helpful: being notified whenever a value changes lets the environment detect what has been added/deleted and then proceed to send a suitable triggering event (**AddBelief/RemoveBelief**) to its agents. This is how agents' *perception* work at the ground level.

Kotlin-BDI provide a certain degree of customisation for environments:

- Environment can be *open* or *closed*, where the latter means that the environment does not let the users add agents to it at run time.
- Users can specify how the environment passively reacts to incoming messages by defining some callbacks.
- Users can specify some *propagation conditions* on messages. These are rules (ie. predicates) that must be tested against any incoming message. If a message passes every defined condition it will be effectively managed and propagated by the environment.

4.4 DSL

One of the main advantages of using the library is that it aims to be as easy to use as possible. *Kotlin-BDI* provides a convenient DSL (domain-specific language) to write agents and environments in a natural way, hiding any kind of complexity away from the users. Kotlin's language flexibility makes the creation of custom DSLs really straightforward using inline functions and the **@DslMarker** annotation, which enables scope/context management - a key concept in any programming language - out of the box with a single line of code.

4.4.1 Environment

An empty environment can be easily created with just one line of code:

```
1 val openEnvironment = open environment {}
2 val closedEnvironment = closed environment {}
3 val environment = environment {}    // Open by default
```

Agents (whose configuration will be covered in the next subsection) can be added inside the environment scopes in different ways:

```

1 val environment = open environment {
2     agent("alice") {}    // Standard creation
3     agent("bob")
4     "carl" with {}       // Extension function on Strings
5 }

```

Environments can be created with some initial `EnvironmentData`), modelled as a map or a list of key-value pairs:

```

1 val environment = open environment {
2     data {
3         "name" with "Carl"
4         "number" with 4
5         pair("bool", true)
6         pair(pairOf("list", kotlin.collections listOf(1, 2, 3)))
7         "term" with structOf("struct", "atom")
8     }
9 }

```

Environments can be enhanced with `Artifacts`, which will be added to the environments' `Workspace`. An artifact method may be called as a result of the execution of an agent's external action:

```

1 val environment = open environment {
2     artifact(object: Artifact {
3         override val id = "firstArtifact"
4
5         fun debugMessage(e: Environment, sender: Agent, message: Term) {
6             val senderName = sender.name
7             val content = message.toTrimmedString()
8             println("Received message ${content})} form ${senderName}")
9         }
10    })
11    artifacts(object: Artifact {           // Multiple insertion at once
12        override val id = "secondArtifact"
13    }, object: Artifact {
14        override val id = "thirdArtifact"
15    })
16 }

```

Users can also control whether a message should be propagated or not and they can also specify how the environment should react to them:

```
1 val environment = open environment {
2     // Propagation condition: no shouts allowed!
3     pass { message -> message.messageAction != MessageAction.Shout }
4
5     // Print the message structure when received
6     onMessage { env, message ->
7         println("Message through the environment: ${message}")
8     }
9 }
```

4.4.2 Agent

Agents' dispatcher, reasoning procedure and execution/termination policy can be all easily specified by assigning them to their relative variable. If not explicitly defined, agents will be created with default conditions.

```
1 val alice = agent("alice") {
2     dispatcher = Dispatcher.PeriodicDispatcher(Duration.ofMillis(500))
3     executionPolicy = ExecutionPolicy.WholeProcedureAtOnce
4     terminationPolicy = TerminationPolicy.ContinueIndefinitely
5     reasoningProcedure = ReasoningProcedureFactory.getDefaultFSMProcedure()
6 }
```

The starting configuration (or state) of an agent can be defined specifying the initial belief base, events and plans.

Here some convenient example regarding some different ways to define beliefs:

```

1  val alice = agent("alice") {
2      beliefs {
3          belief("position"("north"))
4          belief { "position"("west") }
5          belief {
6              val s1 = "door"("left")
7              val s2 = "window"("right")
8              s1 and s2 // Composition of Structs
9          }
10         belief("cat")
11         // Simpler way using an extension method for Strings
12         +"location"
13         +"room"()
14         +"light"("on")
15     }
16     beliefs {
17         +structOf("room", "bedroom", "north")
18     }
19 }

```

Events defined during the setup of the agent can be extremely useful to immediately trigger a plan execution (ie. an agent starting plan).

```

1  val alice = agent("alice") {
2      events {
3          +achieve("find"("treasure"))
4          -achieve("hide"("treasure"))
5      }
6      events {
7          +test("is"("human"))
8          -test("has"("body"))
9      }
10     events {
11         +"light"("on")
12         -"light"("on")
13     }
14 }

```

The DSL makes the creation of plans easier by providing a natural way to express the triggering event, the context and the body formulae:

```

1  val alice = agent("alice") {
2      // Combination of empty plans related to different triggering events
3      // Achievement
4      plans {
5          +achieve("location"("robot")) then {}
6          -achieve("location"("X")) then {}
7      }
8      // Test
9      plans {
10         +test("light"("on")) then {}
11         -test("light"("X")) then {}
12     }
13     // Belief
14     plans {
15         +"room"("bedroom") then {}
16         -"room"("X") then {}
17     }
18 }

```

```

1  val alice = agent("alice") {
2      // Plans can be guarded with certain composable conditions
3      plans {
4          +achieve("move"("X")) guard "robot"("Y") then {}
5          +achieve("move"("X")) guard "robot"("Y") and "path"("Y", "X") then {}
6          +achieve("move"("X")) guard { "robot"("Y") and "path"("Y", "X") } then {}
7          +achieve("move"("X")) guard {
8              val g1 = "robot"("Y")
9              val g2 = "path"("Y", "X")
10             g1 and g2
11         } then {}
12     }
13 }

```

```

1  val alice = agent("alice") {
2      plans {
3          +achieve("known"("robot")) then {
4              // Achievement goal addition/deletion
5              +achieve("move"("X", "Y"))
6              -achieve("move"("X", "Y"))
7              // Test goal addition/deletion
8              +test("location"("Z"))
9              -test("location"("Z"))
10             // Belief addition/deletion
11             +"location"("robot", "bedroom")
12             -"location"("robot", "kitchen")
13             // Parallel achievement goal addition/deletion
14             +parallel(achieve("move"("Z")))
15             -parallel(achieve("move"("Z")))
16             +parallel(test("move"("Z")))
17             -parallel(test("move"("Z")))
18             // Actions, both internal and external
19             internalAction { println("Running action"); it } // Context-free
20             internalAction("print"("hello"))                // Plugin method
21             action("save"("X"))                               // Artifact method
22         }
23     }
24 }

```

Conceptually similar to **Artifacts** and **Workspace**, users can add **Plugins** an agent's **PluginLibrary** to enhance its capabilities. Their method can be called used the **internalAction** formula (while the **action** one will ask the environment to execute an **Artifact** method).

```

1  val environment = open environment {
2      agent("alice) {
3          events {
4              +achieve("printSelf")
5          }
6          plans {
7              +achieve("printSelf") then {
8                  // Calling the plugin method
9                  internalAction("print", atomOf("message"))
10                 // Specifying the plugin name
11                 internalAction("printer", "print", atomOf("message"))
12                 // By using the corresponding structs
13                 internalAction("print"("message"))
14                 internalAction("plugin"("printer","print"("message")))
15                 // Asking the execution of the artifact method
16                 action("debug","agentMessage")
17                 action("debugger","debug","agentMessage")
18                 action("debug"("agentMessage"))
19                 action("artifact",("debugger", "debug"("agentMessage")))
20             }
21         }
22         plugin(object: Plugin {
23             override val id = "printer"
24             fun print(agent: Agent, messageAtom: Atom) {
25                 println("Print: ${messageAtom.toTrimmedString()}")
26             }
27         })
28     }
29     artifact(object: Artifact {
30         override val id = "debugger"
31         fun debug(e: Environment, sender: Agent, message: Term) {
32             val senderName = sender.name
33             val content = message.toTrimmedString()
34             println("Received message ${content})} form ${senderName}")
35         }
36     })
37 }

```

5 Validation

The project has been validated all throughout the development by writing test classes with the JUnit 5 library. Test plans have been organised since the beginning of the project into nested subclasses (marked with the `@Nested` annotation) containing methods with clear method names for easier recognition (thanks to Kotlin backtick quotes enclosing arbitrary strings¹).

At the moment of writing, *Kotlin-BDI* test classes are composed by 98 test methods, divided between the two modules as such:

- Module `bdi-agents` (73 test methods):
 - `Test2pKtIntegration.kt`
 - `TestLogicStructures.kt`
 - `TestAgentSetup.kt`
 - `TestAgentReasoning.kt`
 - `TestReasoningProcedure.kt`
 - `TestEnvironment.kt`
- Module `bdi-agents-dsl` (25 test methods)
 - `TestDSLAgent.kt`
 - `TestDSLEnvironment.kt`

6 Deployment instructions and examples

Kotlin-BDI has been originally developed on top of the PIKA *Coordination* project as a separate and independent branch. The two modules follow the original project standard of using Gradle as the compile and build tool, so users should be able to import the library easily as a dependency in their own project's `build.gradle` or `build.gradle.kts` file.

Here is an example of communication between two agents, modelling the classing “ping-pong” exchange, and one showing how we can create plugins to define new features. As you can see in the latter example, the plugin method returns a `Subscription`, which will then be used to unify to the remaining formulae of the plan body.

¹[Kotlin code conventions for test methods names](#)

```

1  val environment = open environment {
2      artifact(object: Artifact {
3          override val id = "pingPongCounter"
4
5          // The returned EnvironmentData will update the environment state,
6          // which will then send a triggering event to the agents
7          fun track(e: Environment, sender: Agent, pingOrPong: Atom)
8              : EnvironmentData {
9              val data = e.data
10             val event = pingOrPong.toTrimmedString()
11             // Get ping or pong value
12             val currentValue = data[event] as? Int ?: 0
13             // Update data by one and return a new EnvironmentData
14             return EnvironmentData(data + pairOf(event, currentValue + 1))
15         }
16     })
17     // Alice
18     agent("alice") {
19         dispatcher = Dispatcher.PeriodicDispatcher(Duration.ofMillis(500))
20         events {
21             +"pong"(0) // Initial event to trigger the first communication
22         }
23         plans {
24             // The agent will send a "Perform" event to the environment
25             +"pong"("X") then {
26                 internalAction { perform("track"("ping")); it }
27             }
28         }
29     }
30     // Bob
31     agent("bob") {
32         dispatcher = Dispatcher.PeriodicDispatcher(Duration.ofMillis(500))
33         plans {
34             // Same as alice, but with an external action
35             +"ping"("X") then {
36                 action("track"("pong"))
37             }
38         }
39     }
40 }
41
42 // Run the agents
43 environment.runAgents()

```

```

1  val agent = agent("agent") {
2      dispatcher = Dispatcher.PeriodicDispatcher(Duration.ofMillis(300))
3      events {
4          +achieve("recursiveSum"(listOf(1, 2, 3, 4, 5), "X")) // X should be 15
5      }
6      plans {
7          +achieve("recursiveSum"("List", "Result")) then {
8              internalAction("recSum"("List", "Result")) // Result = sum(List)
9              + "resultRecursiveSum"("Result") // Updating the belief base
10          }
11      }
12      plugin(object: Plugin {
13          override val id = "calculator"
14
15          /*
16           Recursive sum:
17           listsum([], 0).
18           listsum([Head|Tail], Result) :- listsum(Tail, SumOfTail),
19                                           Result is Head + SumOfTail.
20          */
21          fun recSum(agent: Agent, c: Term, r: Term): Substitution {
22              val cons = c as Cons
23              val result = r as Var
24              // The list is empty (base case)
25              if (cons.isEmptyList) {
26                  return Substitution.of(r, Integer.of(0))
27              }
28              val head = cons.head as Integer
29              val tail = cons.tail as List
30
31              if (tail is EmptyList) {
32                  // The tail is empty, we have one single element
33                  return Substitution.of(r, head)
34              } else {
35                  // Calling the same function on the tale
36                  val resSub = recSum(agent, tail, r)
37                  // Retrieve the result
38                  val listSum = resSub[r] as Integer
39                  // Calculate the sum of the head with the list sum accumulator
40                  val totalSum = Integer.of(head.intValue + listSum.intValue)
41                  // Unify the result with the retrieved sum
42                  return Substitution.of(r, totalSum)
43              }
44          }
45      })
46  }

```

7 Conclusions and future works

The development of *Kotlin-BDI* has now spanned over several months, considering all the different phases as such as the documentation and study of the AgentSpeak language, the analysis on what it meant to solve, the design of the main components and the actual implementation of both the core module and the DSL. What has started as a project for an university course - limited in scope and time - has quickly grown and it was immediately clear to me that the project could gain some real potential of being useful for anybody who works in the field. All considered, I must say I feel quite proud of the final result. But at the same time, I am completely aware there is still lots of work to do and I am really looking forward to adding some features that I decided to temporarily leave behind.

To sum up the most important ones (most of which are present in other AgentSpeak implementations like Jason[9]):

- The library should handle plan failures in a better way: at the moment failed intentions just get cleared from the intentions set and thus there is no way for the developer to handle those situations.
- The core components should be updated to work nicely with any kind of `2pKt Term`, while at the moment the library mainly supports any subclass of `Struct`.
- There is no support for atomic update of a belief (ie. the deletion and insertion in a single formula).
- There is no support for explicit/strong negation, so test goals and assumptions on unknown belief will always result false.

References

- [1] Anand S. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. *7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, 1996.
- [2] Michael P. Georgeff and Amy L. Lansky. Procedural knowledge. *The Proceedings of the Institute of Electrical and Electronics Engineers*, 74(10):1383–1398, 1986.
- [3] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. *6th National Conference on Artificial Intelligence (AAAI-87)*, 1987.
- [4] 2pKt. <https://gitlab.com/pika-lab/tuprolog/2p-in-kotlin>. Accessed: 2021-03.
- [5] tuProlog. <https://apice.unibo.it/xwiki/bin/view/Tuprolog>. Accessed: 2021-03.

- [6] Coordination with TuSoW. <https://gitlab.com/pika-lab/tuples/coordination>. Accessed: 2021-03.
- [7] Álvaro F. Moreira and Rafael H. Bordini. An Operational Semantics for a BDI Agent-Oriented Programming Language. *Proceedings of the Workshop on Logics for Agent-Based Systems*, 2002.
- [8] A. Ricci, M. Viroli, and A. Omicini. “Give Agents their Artifacts”: The A&A Approach for Engineering Working Environments in MAS. *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, 2007.
- [9] Jason. <http://jason.sourceforge.net/wp/>. Accessed: 2021-03.
- [10] Anand S. Rao and Michael P. Georgeff. BDI Agents: From Theory to Practice. *1st International Conference on Multi Agent Systems (ICMAS 1995)*, 1995.