Kotlin-BDI

Autonomous Systems

Marco Galassi

University of Bologna

March 2021

Table of Contents

Goal

Original goal

"Design and prototyping of an AgentSpeak(L) implementation as a Kotlin DSL, in TuSoW"

Goal

Original goal

"Design and prototyping of an AgentSpeak(L) implementation as a Kotlin DSL, in TuSoW"

Revised goal

"The creation of an AgentSpeak(L) abstract language implementation as a Kotlin DSL"

Requirements

- The library must provide a complete and functional implementation of (at least) the basic AgentSpeak language features in Kotlin.
- The library must provide a DSL (domain-specific language) which should be modelled to resemble logic programming and should enforce a natural way to think about agents.

Requirements

- 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.

Table of Contents

Analysis I

Logic programming in Kotlin

Kotlin, despite being a general purpose language, does not support logic programming in itself and, so this requires a tool (a library or a piece of software) to fill the gap.

The concept of *environment*

- The container in which all the agents live?
- The host of anything that can be perceived and capable of change?
- The entity which enables communication between agents?

Analysis II

The concept of *action*

Actions represent the way agents can affect the surrounding environment, possibly changing its state. But how?

Communication between agents

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.

Table of Contents

Structure

Kotlin-BDI systems

The library lets users build systems based on the interaction between environments and agents.

Being designed as independent entities, agents and environments do not strictly depend on one another...

Structure

Kotlin-BDI systems

The library lets users build systems based on the interaction between environments and agents.

Being designed as independent entities, agents and environments do not strictly depend on one another...

...although complex features may require some kind of coordination between them.

Structure - Environments

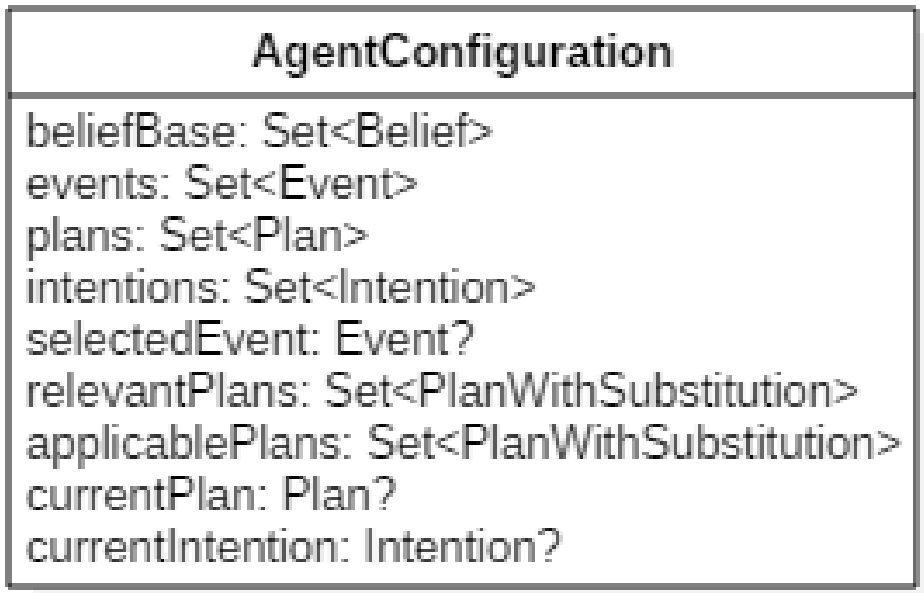
- They are the stage on which multiple agents act, a sort of permanent surrounding. An environment can thus contain one or more agents.
- They keep track of everything that happens outside the private scope of the agents, as a sort of common and shared state which can be perceived and can be changed.
- They enable communication between agents, thus being the underlying channel in which messages travel.
- They can be enhanced with artifacts.

Structure - Agents

- The proactive and autonomous entities of the systems, executing in a separate flow of control.
- Endowed with a non-blocking channel for asynchronous communication.
- They can be enhanced with plugins.

Agent configuration

At any given time, an agent can always be described by a certain state or configuration.



Behaviour

System 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.
- is modelled as a sequence of steps (formally called execution steps) that must be followed.
- can be executed step-by-step or all at once.

The AgentSpeak(L) implementation is basically a specific reasoning procedure with strictly defined steps, modelled after Rao's pseudocode.

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

Interaction I

Agents and their surrounding environment communicate asynchronously by exchanging messages, modelled as logic structures whose semantics is defined by a specific action, through the environment.

message(sender(a1), receiver(a2), action(act), payload(pld))

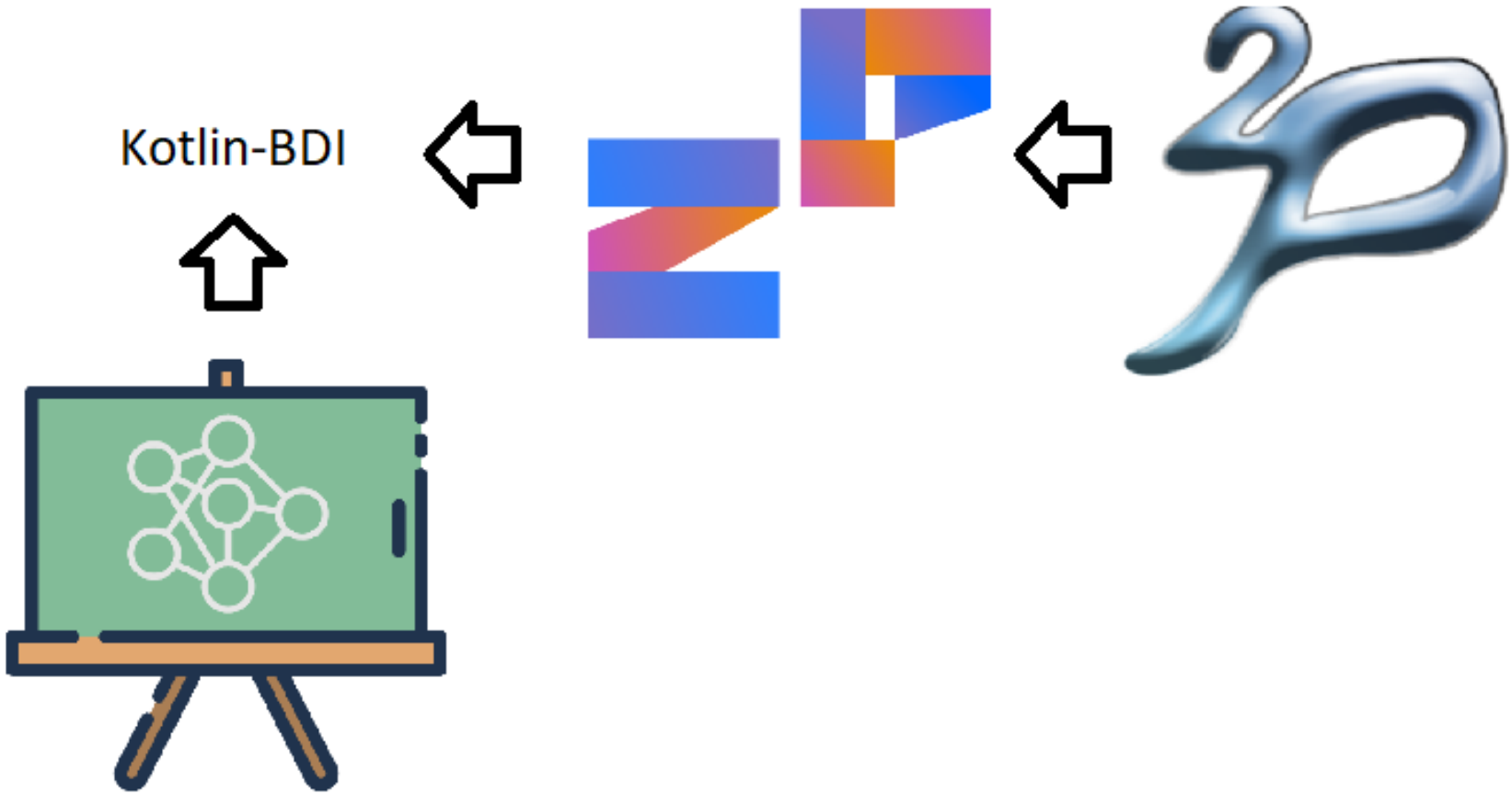
Interaction II

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

Messages are sent when agents want to communicate with each other or when the environment passively notifies the agents of a change in its state.

Table of Contents

Technologies



Dispatcher

Agents' control flow can be customised by declaring a certain dispatcher to be used for them.

By using a specific dispatcher, 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.

Agent `onStep()` method

```
1  //Consider pending events
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 wether to stop or to continue
13 when(terminationPolicy) {
14     TerminationPolicy.StopAfterOneExecution ->
15         controller.stop(newAgentState)
16     TerminationPolicy.ContinueIndefinitely ->
17         controller.`continue`(newAgentState)
18 }
```

AgentSpeak reasoning procedure

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.

`Feature`, `Plugin` and `Artifact`

Kotlin-BDI supports another two kind of actions:

- External actions, which may change the state of the environment, producing effects outside the scope of the calling agent.
- Internal actions, which are completely encapsulated in the agents scope.

Both action types can enhance environments and agents capabilities in a flexible way.

Through Kotlin reflection, users can write anonymous classes implementing the `Feature` interface to define custom methods which can be executed at run-time. Two main `Feature` subclasses:

- `Plugin`, which can be collected in a `PluginLibrary` and linked to an agent to be used during an internal action.
- `Artifact`, which can be collected in a `Workspace` and linked to an environment to be used when one of its agents tries to execute an *external* action (by sending a *perform* message).

Environment state

Environment state is modelled as an instance of `EnvironmentData`, a map pairing `String`s to `Any` values (type handling must be managed by the developers).

This is an observable variable: whenever the environment detects a change, it proceeds to send a suitable triggering event to its agents. This is how agents' perception of the environment works at the ground level.

Table of Contents

Environment

Empty environments created with just a line of code:

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

Agents 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  }
```

Environment - `EnvironmentData`

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  }
```

Environment - `Artifact` and `WorkSpace`

```
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 }
```

Environment - Conditions and reactions

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  }
```

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
6                              .getDefaultFSMProcedure()
7  }
```

Agent - Initial 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 }
```

Agent - Initial events

```
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 }
```

Agent - Plans definition

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

Agent - Guarded plans

```
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 }
```

Agent - Plan body formulae

```
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 }
```

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