ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

JaKtA: BDI Agent-Oriented Programming in Pure Kotlin

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Baiardi M., Burattini S., Ciatto G., Pianini D. (2023). JaKtA: BDI Agent-Oriented Programming in Pure Kotlin. Cham : Springer [10.1007/978-3-031-43264-4_4].

Availability:
This version is available at: https://hdl.handle.net/11585/950535 since: 2023-12-13

Published:
DOI: http://doi.org/10.1007/978-3-031-43264-4_4

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

06 May 2026

This is the final peer-reviewed accepted manuscript of:

Baiardi, M., Burattini, S., Ciatto, G., Pianini, D. (2023). JaKtA: BDI Agent-Oriented Programming in Pure Kotlin. In: Malvone, V., Murano, A. (eds) Multi-Agent Systems. EUMAS 2023. Lecture Notes in Computer Science(), vol 14282. Springer, Cham.

The final published version is available online at: https://doi.org/10.1007/978-3-031-43264-4_4

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/)

When citing, please refer to the published version.

JaKtA: BDI agent-oriented programming in pure Kotlin*

Martina Baiardi[2] [0009-0001-0799-9166], Samuele Burattini[1] [0009-0009-4853-7783], Giovanni Ciatto[1] [0000-0002-1841-8996], and Danilo Pianini[1] [0000-0002-8392-5409]

[1,2] Department of Computer Science and Engineering (DISI)
Alma Mater Studiorum—Univerisità di Bologna
Via dell'Università 50, 47522 Cesena (FC), Italy
[1] ⟨name⟩.⟨surname⟩@unibo.it,
https://www.unibo.it/sitoweb/⟨name⟩.⟨surname⟩/en
[2] m.baiardi@unibo.it, https://www.unibo.it/sitoweb/m.baiardi/en

Abstract. Multi-paradigm languages are becoming more and more popular, as they allow developers to choose the most suitable paradigm for each task. Most commonly, we observe the combination of object-oriented (OOP) and functional programming (FP), however, in principle, other paradigms could be hybridised. In this paper, we present JaKtA, an internal DSL adding support for the definition of belief-desire-intention (BDI) agents in Kotlin. We believe is a first step to investigate the blending of Agent-Oriented Programming (AOP) with other popular paradigms and we discuss the opportunity and value of doing so with an internal DSLs. Finally, through JaKtA, we show how this can already lead to compactly and expressively create BDI agents that smoothly interoperate with the host language, its libraries and tooling.

Keywords: BDI · AgentSpeak(L) · DSL · Kotlin · JaKtA

1 Introduction

Many modern mainstream programming languages natively support multiple programming paradigms, thus allowing programmers to use the most appropriate abstractions for the job at hand without the need to adapt their mind to a syntax and tooling different to the one they are acquainted with. Most frequently, we observe the combination of object-oriented (OOP) and functional programming (FP) paradigms: some notable examples are OCaml [16], which adds object-orientation on top of the functional paradigm; Java, that since version 8 supports some functional abstractions on top of OOP [17] via the *lambda expressions* and the *stream API*; and Scala, that since its conception has been designed with both OOP and FP in mind [24].

* This work has been partially supported by the Chist-Era IV project "Expectation", and by the Italian Ministry for Universities and Research (G.A. CHIST-ERA-19-XAI-005).

To the best of our knowledge, however, no mainstream programming language currently features *native* support for the agent-oriented programming paradigm (AOP), especially the beliefs–desires-intentions (BDI) model. The current state of the art includes several stand-alone programming languages that support BDI agents programming following the well-known AgentSpeak(L) [22] semantics—such as Jason [3], Astra [9], and GOAL [13]. However, using and maintaining stand-alone languages can be burdening, especially when the community of contributors is small, since languages usually require several tools to be usable in practice (e.g., content assistants, syntax highlighters, linters, checkers, debuggers, etc.) whose development and maintenance adds upon the cost of the language itself—potentially causing the ecosystem to evolve slowly, and thus hindering adoption.

In this paper, we propose a solution to both the availability in the mainstream and the tooling support of BDI languages, by leveraging a recent trend in modern programming languages: the construction of *internal* domain-specific languages (DSLs), namely, carefully designed APIs that capture problem-specific abstractions into a syntax providing ergonomics akin to that of a dedicated language, but still letting users rely on the tooling and ecosystem of the host language, as well as transparently use abstractions from other paradigms on a per-need basis. Thus, inspired by the successful Jason AOP language, we present Jason-like Kotlin Agents (JaKtA): a Kotlin internal DSL meant to seamlessly integrate BDI agents into a mainstream programming language, adding AOP to Kotlin as an additional paradigm, retaining its toolchain, libraries, and OOP/FP abstractions. We show that the internal DSL approach can blur the (usually neat) boundary between the two paradigms, promoting a more natural and seamless interaction. Moreover, since the code using the DSL abstractions is still valid code in the host language, we show that the tooling of the host language can be used immediately, with no need for additional support software to be developed and maintained.

The remainder of this paper is organised as follows: in Section 2, we present DSL engineering and we summarise the state of the art of BDI languages, then in Section 3 we discuss the design and the main features of JaKtA, and we show how it can be used to compactly and expressively create BDI agents that smoothly interoperate with the Kotlin ecosystem; in Section 4 we assess the effectiveness of our internal DSL approach by showing, through practical examples, how it can simplify the development of BDI agents in some conditions; and finally, in Section 5, we conclude the paper by discussing some limitations of our approach, as well as some future research directions stemming from it.

2 Background

This work lays on two pillars: DSL engineering (specifically, internal Kotlin DSLs) and BDI agents programming. In this section, we briefly introduce them by discussing the principles behind the creation of DSLs and we explain how and why modern languages support the creation of *internal* DSLs. We also provide

a comparison among existing BDI programming frameworks from the literature, discussing how syntactical aspects may impact their interoperability and versatility.

2.1 DSL engineering

As introduced in Section 1, DSLs are programming languages tailored to specific domains: they expose the domain model entities and their interactions as first-level abstractions. However, there is no rule on which amount of domain-specificity makes a language a DSL: at some level, every language is domain-specific, with the specific domain being the *paradigm* the language is rooted in. For instance, we argue that even the Agent Speak Language (ASL) can be seen as a DSL modelling the domain of BDI agents.

From a technical perspective, DSLs can be classified into two broad categories [24]: *external*, if they are stand-alone, with their own custom syntax and compiler/interpreter; and *internal*, if they are embedded in a host language and rely on the syntactic and semantic features of the host. From the point of view of the host language, internal DSLs are indistinguishable from ordinary libraries (indeed, as C++ inventor Bjarne Stroustrup used to say, "library design is language design" [25]), their distinction is usually driven by their *purpose*[1]. Consequently, internal DSLs might *in principle* be realised in any language; *in practice*, however, the host language syntactic flexibility directly reflects on the ergonomics of any internal DSL. For this reason, several recent languages (e.g., Scala, Kotlin, Ruby) provide syntactic features specifically tailored to the constructions of internal DSLs. Despite these features simplify the adoption of internal DSLs, they cannot provide the same expressiveness of an external DSL, as they are still bound to the host language syntax, for example, in the case of Kotlin each DSL statement must be enclosed in a curly braces block.

Selecting whether an internal or external DSL is best for the problem at hand is a matter of trade-offs: as discussed, internal DSLs have limited syntactic flexibility that could result in a less expressive language, but, in turn, they inherit from their host: *(i)* the tooling (IDE support, build systems, linters, debuggers, profilers, and so on), reducing the maintenance burden; *(ii)* the libraries, reducing the need for ad-hoc solutions; and *(iii)* the abstractions, allowing the DSL to be used in conjunction with other paradigms. Together, these aspects may also lower the learning curve for those already acquainted with the host language, possibly favouring wider adoption.

2.2 BDI paradigm and programming languages

The philosopher Michael Bratman described humans' practical reasoning via the "beliefs, desires, intentions" (BDI) framework, as a way to explain future-directed decision-making [4]. Successively, the framework was formalised by means of modal logics [8], and then turned into an abstract semantics for computational

[1] https://www.martinfowler.com/bliki/DslBoundary.html

Table 1: Comparison of the identified practical features across several common BDI agent programming languages. Columns denote languages, rows denote features. JaKtA is the language proposed in this paper: it is reported here for to ease comparison. In non-textual cells, symbol ✓ indicates the feature availability, × unavailability, and ∼ that we were not able to find conclusive evidence.

	JaKtA	**Jason [3]**	**Spade-BDI [20]**	**Phidias [10]**	**Astra [9]**	**JACK [26]**	**Jadex [21]**	**Goal [13]**
DSL Type	internal	external	both	internal	external	external	external	external
Hosting Syntax	Kotlin	AgentSpeak(L) extension	Python	Python	custom Java extension	custom Java extension	XML Java annotations	custom Prolog extension
Execution Platform	JVM	JVM	Python	Python	JVM	JVM	JVM	JVM
Direct interop.	Any JVM language	Any JVM language	Python	Python	Any JVM language	Any JVM language	Any JVM language	SWI-Prolog
Paradigm blending	✓	×	✓	✓	✓	✓	×	✓
Type safety	✓	×	×	×	✓	✓	✓	×
Reuse mechanisms	Any Kotlin mechanism	file incl., ext. actions	Any Python mechanism	Any Python mechanism	agent extension	reusable plans	selective file incl.	reusable plans, beliefs, goals, and agents
Logic Programming	✓	✓	×	✓	×	×	×	✓
License	Apache 2.0	LGPL v3	GPL v3	MIT	GPL v3	Proprietary	GPL v3	GPL v3

agents: AgentSpeak(L) [22]. Computational agents are *autonomous* entities [19] situated into an *environment* they can perceive and affect; they interact either directly or stigmergically through the environment [23]. The classical implementation of BDI agents, based on the Procedural Reasoning System (PRS) [12], is characterized by four main abstractions, namely: **beliefs:** a set of facts and rules constituting the agent's *epistemic* memory; **desires:** a set of goals, (possibly partial) descriptions of the states of the world the agent wants to achieve, test, or maintain; **intentions:** a set of tasks the agent is currently committed to; **plans:** a set of *recipes* representing the agent's procedural memory.

Since its introduction, the community produced many programming languages for BDI agents. Most of them are either based on or inspired by the AgentSpeak(L) semantics. In this section, we compare several major BDI agent programming languages from a software engineering perspective. Details about the comparison are reported in table 1. There, columns represent BDI languages, while rows represent features that those languages may (or may not) have.

As far as BDI agent programming languages are concerned, our comparison is focussing on those languages which appear to have some running software implementation which is actively maintained and used by the community. Hence, we build upon the recent work by Calegari *et al.* [5], which surveys the state-of-the-art of logic-based agent-oriented technologies, and we select the ones aimed at supporting general-purpose BDI agents programming.

Conversely, as far as features are concerned, in the remainder of this section we discuss the most relevant ones, namely: *(i)* **DSL type** (internal or external); *(ii)* **hosting syntax**, i.e., which syntax the DSL is embedded in (for internal DSLs) or based upon (for external DSLs); *(iii)* **execution platform**, i.e. which runtime platform the language runs upon; *(iv)* **direct interop**erability, i.e., whether other languages can be called from within the BDI language (and, in that case, which ones); *(v)* **paradigm blending**, i.e., whether it is possible to mix, in the same source and scope, AOP and other abstractions; *(vi)* **type**

safety i.e., the ability of the compiler/interpreter to intercept (most) type errors ahead-of-execution; *(vii)* **reuse mechanisms**, i.e., whether and how it is possible to parameterise and reuse partial or entire MAS specifications; *(viii)* **logic programming** support, i.e. the capability to rely upon the mechanisms of unification and backtracking to represent and manipulate BDI data structures; and, finally, *(ix)* **license**.

DSL type and hosting syntax. The former feature categorizes BDI languages as either *external* or *internal* DSL, or possibly both of them. Conversely, the latter feature provides further details about the DSL syntax. The two features are strictly related, as they both refer to the syntax of the language. In fact, for internal DSLs, one may be interested in understading which syntax the DSL is embedded in, whereas, for external DSLs, we further describe the derivation of the syntax. Accordingly, for internal DSLs, the hosting syntax is quite straightforward: both Spade-BDI and Phidias are hosted by Python. Conversely, external DSLs' syntaxes are built as extensions or refinements of well-known languages. For instance, while Jadex relies on XML, Goal extends Prolog [15], and Jason extends AgentSpeak(L); whereas Astra and JACK extend Java.

Execution platform. The execution platform is the runtime environment which is required for running a given BDI language—as well as the MAS described through it. It is worth highlighting that several programming languages may be executed on the same platform. This is the case, for instance, in Kotlin, Java, and Scala which are all executed on the JVM platform. The execution platform is a relevant feature, as it may affect the portability of the MAS, as well as its interoperability with other systems and languages. Accordingly, while Spade-BDI and Phidias target the Python platform, the other languages target the JVM platform.

Direct interoperability. This feature concerns the ability of the agent programming language to interact with the hosting language constructs. Specifically, this feature is about which other languages the BDI language at hand can directly call, exploiting the hosting language interoperability mechanisms. For instance, every language targetting the JVM can directly call all the other JVM languages. However, this is not the case for Jadex, which is implemented on Java, but exploits XML files for MAS specification.

Paradigm blending. This is a *syntactical* feature of languages whose syntax mixes AOP constructs with the hosting language ones—for instance, by letting developers exploit both AOP and OOP constructs, if the hosting language is OO. Notice that the opposite situation may also occur. In fact, some languages enforce a clear separation among high-level AOP constructs (e.g. belief, goals, plans) and the hosting language ones (e.g. classes, functions, etc.). This separation may for instance be enforced by requiring the AOP portions of a MAS to be written in separate files. For instance, in Python-based BDI languages such as Spade-BDI and Phidias, AOP specifications consist of Python classes and

methods. Conversely, Astra, JACK, and Goal allow exploiting Java or Prolog libraries, respectively. Finally, Jason, and Jadex strongly separate AOP from OOP. There, MAS are composed by scripts describing agent specifications, and by actions/environment specifications. The former only support AgentSpeak(L)-compliant constructs, whereas the latter are ordinary Java code.

Type safety. This feature refers to the presence of a strong type checker for the BDI language at hand, which may proof check agents specifications at compile-time. Solutions having a tight interoperability with Java, such as Astra, JACK, and Jadex, come with this feature; whereas the others do not. Other languages – such as Jason, Spade-BDI, Phidias, and Goal – come with a more flexible syntax—as they rely on weakly-typed hosting languages such as Prolog or Python.

Reuse mechanisms. This feature refers the presence of abstraction mechanisms supporting the reuse of partial MAS specifications. As far as this feature is concerned, we observe great variety among the surveyed languages. Some rely on bare file include mechanisms. This is the case, for instance, of Jason – which supports the inclusion of ASL files into other ASL files, by path –, and Jadex—which supports referencing XML or Java files into other XML files. Furthermore, virtually all surveyed solutions support the abstraction and reuse mechanisms of the hosting language, if any. This implies, for instance, that solutions based on an OO hosting language may take advantage of OOP abstraction mechanisms such as sub-typing and inheritance for the OO portions of their MAS specifications. Some solutions may also expose high-level, agent-oriented notions – such as agents or plans – as first-class syntactical constructs. In other words, they may support ad-hoc syntaxes for writing agents or plans. This is the case, for instance, of Astra, JACK, and Goal. When this is the case, first-class abstraction can be re-used along the MAS specification. For example, Astra supports writing agents specifications extending other agents specifications.

Logic programming support. This feature is about whether BDI languages rely on full-fledged logic programming as the preferred means to represent and manipulate BDI data structures—e.g. beliefs, goals, etc. This is the case, for instance, of Jason, Phidias, and Goal, which use logic terms and clauses to represent beliefs, goals, plans, and events. They also rely on logic unification and resolution as the basic mechanism to manipulate these data structures to implement the BDI reasoning cycle.

License. This feature is about which license BDI solutions are distributed with. Notably, most solutions come with an open-source license, and their source code is freely available and inspectable on the Web. The only exceptions are JACK, which is proprietary, closed-source software, and Goal which allegedly has an open-source license, despite we were not able to find the source code on the Web.

3 A Kotlin DSL for BDI Agents

We now let the analysis from Section 2.2 drive our selection of core features, that will lead, in turn, to the actual implementation of a DSL for BDI agents.

We want our DSL to be familiar for BDI experts and, at the same time, to look idiomatic to the community of mainstream developers. One way to achieve the first goal is to reach AgentSpeak(L) compliance (i.e., to fulfill its operational semantic), as AgentSpeak(L)-inspired languages are very popular within the AOP community. Concerning the second goal, it can be achieved by letting the DSL (and the underlying agent interpreter) be compliant with the API, the syntax, and the stylistic conventions of some mainstream language of choice. Programmers from both communities must be able to blend paradigms, writing pieces of code that mix BDI abstractions with the ones of the chosen mainstream language. Together with the will to inherit the existing reuse mechanisms of a mainstream language, these aspects led us to choose an *internal* DSL.

As a BDI agent programming language, we also require our DSL to be compliant w.r.t. a set of features, discussed below. First, the language should support strong typing, and possibly type inference, in order to keep types as hidden as possible. It should also support modularity and reusability at various levels, there including *(i)* agent specifications, *(ii)* plan libraries and/or individual plans, *(iii)* belief bases or goal sets, as well as *(iv)* internal and external actions. This implies all such syntactical categories could be in principle written in separate files and composed in the finest way possible. Writing all such categories in a single file should be supported as well.

The DSL should support an explicit notion of *environment*, which in turns supports the pluggability of custom *external* actions – i.e., custom functionalities that agents may invoke to support perception and actuation – as well as the pluggability of custom message passing mechanisms—hence virtually supporting distributed communication among agents. As far as pluggability is concerned, MAS specification written in JaKtA should also support the addition of custom *internal actions* on individual agents – i.e., custom functionalities supporting the inspection/modification of agents' internals –, as well as the choice of the most adequate concurrency model for the MAS at hand—i.e., roughly, the strategy by which agents' concurrent execution is scheduled by the OS.

Finally, the DSL should support full-fledged logic programming syntax and semantics in dealing with BDI data structures representation and manipulation.

A more nuanced pick is the selection of the target host language. There are several elements to consider, including the target platform and its portability across multiple platforms (as we want to maximise the range of potential target runtimes), the existing ecosystem (as we want to leverage existing libraries and tools), the language's popularity (as we want to let the agent-orientation be available to the widest possible audience), the type safety, and, of course, the specific language features that could be leveraged for the construction of a DSL.

We considered several languages, including Java, Scala, Kotlin, Python, Ruby, C#, and Typescript. From the point of view of syntactic flexibility we favored Scala, Kotlin, and Ruby, as they provide machinery specifically meant to allow

the construction of DSLs. We then discarded Ruby, as we wanted a statically typed language. We picked Kotlin over Scala despite the latter having a better type system (supporting, for instance, path-dependent and higher-kinded types [14]) for merely practical reasons: *(i)* Scala 3 recently broke retro-compatibility with Scala 2, and, at the time this work was realised, many libraries and tools were not yet available for the new version; *(ii)* we expect Kotlin popularity to grow faster than Scala's in the future, as Google picked Kotlin as reference language for the Android ecosystem[2], and *(iii)* there are emerging libraries in Kotlin that are meant for data science, e.g.: *KotlinGrad*[3], *KMath* [18], *KotlinDL*[4], and *Kotlin Dataframe*[5]. Combined with a Kotlin-based solution for MAS, these tools may hopefully pave the way towards the combination of MAS and data science.

3.1 Architecture and Implementation Details

It is worth mentioning that some required features are not merely syntactical as they require support from the underlying BDI agent interpreter. This is the case, for instance, of features supporting the pluggability of custom message passing mechanisms as well as the choice of the most adequate concurrency model for the MAS at hand. For this reason, JaKtA comes with its own BDI execution engine. Designing from scratch required significant effort, but it also allowed us to decouple agent specifications and their execution, and opened to the possibility to target multiple platforms by leveraging the Kotlin capability to do so.

The JaKtA framework then includes three main modules, namely: *(i)* the JaKtA DSL, *(ii)* the JaKtA BDI interpreter, and *(iii)* the concurrency management module. Notably, the DSL is built on top of the BDI interpreter, which in turn is built on top of the concurrency management module.

In principle, other languages could reuse the BDI interpreter by replacing the DSL module. For instance, a Jason's parser or a new Scala internal DSL for AOP could be plugged on top of the existing BDI interpreter, enjoying, respectively, the Kotlin debug tools and a reduced implementation effort.

The concurrency management module defines how agents are coupled with threads, allowing the same specification to be executed on one or more threads, depending on the application at hand. However, because of space limitations, in the remainder of this paper we focus upon the syntactical aspects of JaKtA, leaving the discussion of the underlying interpreter and concurrency module – as well as the challenges and the opportunities they bring – to future works.

The framework has been released, free and open-source. It is available on GitHub[6] and Maven Central[7] and archived on Zenodo [2].

---

[2] https://developer.android.com/kotlin/first
[3] https://github.com/breandan/kotlingrad
[4] https://github.com/Kotlin/kotlindl
[5] https://github.com/Kotlin/dataframe
[6] https://github.com/jakta-bdi/jakta
[7] https://search.maven.org/artifact/it.unibo.jakta/jakta-dsl

3.2 JaKtA's syntax

JaKtA DSL syntax is strongly inspired by Jason and it is AgentSpeak(L)-compliant. The entry point is the `mas` block, inside whose scope all the elements composing a BDI MAS can be defined:

```
mas { environment { ... }; agent("jedi") { ... }; agent("sith") { ... } }
```

In the `environment` block, users define the external actions that agents can use, as well as what agents can perceive. External actions include communication primitives that can be implemented to send messages of a predefined type to agent message boxes that are reified as part of the environment. Achieving compliance with different agent communication languages (e.g. KQML [11] as used in Jason) requires a further definition of the types of messages an agent can send and how such types are interpreted in the agent lifecycle.

```
mas { environment {
  actions { // definition of the external actions for this environment
    action(create, ...) { addAgent(...) }
    action(talk, ...) { sendMessage(recipient, ...) }
  }
}}
```

Agents are named entities created with the `agent` function. These few syntactic elements are enough to show a hint of how blending paradigms can be leveraged to build complex systems in a few lines of code. In the following example, we mix OOP, FP, and AOP: we fetch the rooster of three Italian football teams from a public website, we extract the names through a regular expression, and then we create one agent for each player:

```
mas { // BDI specification
  fun allPlayers(team: String) = // Object-oriented style
    Regex("""<span class="card-title">((\w+|\s)+)<\/span>""").findAll(
      URL("https://analytics.soccerment.com/en/team/$team").readText()
    ).map { it.groupValues[1] } // Monadic manipulation (functional)
  listOf("napoli", "milan", "juventus")
    .flatMap(::allPlayers) // Functional style (higher-order function)
    .forEach { agent("$player playing for $team") { ... }/* BDI style */}
}
```

In this example, we exploit JaKtA for the MAS definition, the OOP paradigm to deal with the regular expression match and data extraction from the group, and the functional paradigm to monadically map teams to players.

Agents' body is a collection of `beliefs`, `goals`, internal `actions` and `plans` defined in homonym blocks. Beliefs are represented as a logic theory, namely a collection of *facts* and *rules* expressed in a logic programming fashion. JaKtA directly leverages, and exposes as API, the logic programming toolkit for Kotlin 2P-Kt [6] and its internal DSL for Prolog [7].

For instance, in the following, we define one fact (zero is a natural number) and a logic rule defining the 'successor' relation among natural numbers:

```
mas { agent("gauss") { beliefs {
  fact { natural_number(zero) }
  rule { natural_number(successor(X)) impliedBy natural_number(X) }
}}}
```

Goals can indicate either something that the agent wants to `achieve` or something that it wants to `test` (discover). Test goals prioritize the consultation of the knowledge base over the execution of plans.

```
mas { agent("player1") { goals { achieve(victory(X)); test(has_won(Y)) } } }
```

Internal actions can access and modify the agent's state. In the following snippet, an internal action is used to modify the knowledge base of an agent, changing the team it cheers for:

```
mas { agent("turncoat fan") { actions {
  action(changeTeam, 1 /*this parameter is the arity*/) {
    removeBelief(cheeringFor(X))
    addBelief(cheeringFor(argument(0) /*positional access to parameters*/))
  }
}}}
```

Finally, plans describe which operations the agent is capable to perform; inheriting the successful model of Jason, in JaKtA they are composed of a **triggering event** deciding whether the plan is *relevant*, an optional **context** restricting its *applicability*, and a **body** with the implementation. The triggering event can be a goal/belief invocation/addition (+) or failure/deletion (-), in the form: `[+|-]<triggering event> onlyIf {<context>} then {<body>}`. If a logical expression is present in the context block (prefixed by `onlyIf`), it is then used to vet the relevant plan; and if the plan is selected for execution the sequence of operations and actions contained in its body (prefixed by `then`) is performed. In the following example, we showcase the expressivity of blended paradigms by creating a Kotlin function using AOP in JaKtA to verify the Collatz conjecture [1] for a given number:

```
fun collatz(number: Int) = mas { agent(collatz) {
  goals { achieve(collatz(number)) }
  plans {
    +achieve(verify(X)) // We reached 4 for the second time: it's a cycle
      .onlyIf { found(4).fromSelf }
      .then { Print("Collatz Conjecture verified!"); execute(stop) }
    +achieve(collatz(X)) // We reached an even number: divide by 2
      .onlyIf { X.isEven() and (R `is` X.intDiv(2)) }
      .then { achieve(verify(R), true); +found(X); achieve(collatz(R)) }
    +achieve(collatz(X)) // We reached an odd number: multiply by 3 and add 1
      .onlyIf { X.isOdd() and (R `is` ((X * 3) + 1)) }
      .then { achieve(verify(R), true); +found(X); achieve(collatz(R)) }
  }
}}}
```

4 JaKtA in practice: running example

In this section, we show how JaKtA compares with a reference AOP technology (Jason) through a running example in terms of *(i)* multi-paradigm integration, and meta-programming, *(ii)* abstraction, re-use and type safety; and *(iii)* tooling and ecosystem. The case we select is meant to highlight the benefits of paradigm blending: we want to write a multi-agent modelling a TicTacToe match played on a $N \times N$ board, where N is only known at runtime. For the sake of conciseness,

we keep the example deliberately minimal, and we only report the code of a single player. The full code of the example is available on a public repository[8].

The agent may perceive the environment (the board) via percepts of the form `cell(X,Y, Z)`, where `X` and `Y` are the coordinates of the cell, and `Z` $\in$ {`e`, `x`, `o`} is the symbol contained in the cell. The agent may also perceive the beginning of a turn via the `turn(x)` (resp., `turn(o)`) percept, and may place a symbol in a cell of the environment using the `put(X,Y, Z)` *external* action—which also passes the turn. The agent's play strategy is the following: *(i)* if there are N of your (resp. the other player's) marks aligned in a row, declare victory (resp. defeat); *(ii)* if there are $N-1$ of your (resp. the other player's) marks aligned and the Nth cell in the same direction is empty, write your mark in that cell; *(iii)* put a cross in random empty cell.

There are four alignement directions, so the agent's belief base can host:

```
aligned(Cells) :- vertical(Cells) | horizontal(Cells) | diagonal(Cells) |
    antidiagonal(Cells).
```

The critical part of the scenario, however, is dealing with a grid of *unknown size*. For a simple 3×3 case, the problem can be dealt with via four couples of rules in the form:

```
⟨alignment⟩([cell(X, Y, S)]) :- cell(X, Y, S).
⟨alignment⟩([cell(X, Y, S1), cell(A, B, S2) | OtherCells]) :-
    cell(X, Y, S1) & cell(A, B, S2) & A-X=⟨dx⟩ & B-Y=⟨dy⟩ &
    ⟨alignment⟩([cell(A, B, S2) | OtherCells]).
```

where meta-variable $\langle alignment \rangle$ can be: `vertical`, `horizontal`, `diagonal`, and `antidiagonal`, while $\langle dx \rangle$, $\langle dy \rangle$ are in 1, 0, or -1. Under these premises, for a 3×3 simplified scenario, the plans dealing with victory, loss, and random choice may be written in Jason as:

```
+turn(x) : aligned([cell(_,_,x),cell(_,_,x),cell(_,_,x)]) <- .print('I won')
+turn(x) : aligned([cell(_,_,o),cell(_,_,o),cell(_,_,o)]) <- .print('I lost')
+turn(x) : cell(X,Y,e) <- put(X,Y,x)
```

whereas plans making the final move can be written as:

```
+turn(x) : aligned([cell(_,_,x),cell(_,_,x),cell(X,Y,e)]) <- put(X,Y,x)
+turn(x) : aligned([cell(_,_,x),cell(X,Y,e),cell(_,_,x)]) <- put(X,Y,x)
+turn(x) : aligned([cell(X,Y,e),cell(_,_,x),cell(_,_,x)]) <- put(X,Y,x)
```

Plans impeding the victory of the opponent would be very similar.

This way of writing plans, however, does not scale well with the size of the board: a $N \times N$ board would count $2N + 3$ plan statements with a guard mentioning N cells. There are no good strategies to handle these situations in pure Jason (i.e. without using external tools to generate code), while they can be managed by relying on alternative paradigms in JaKtA.

Multi-paradigm integration and meta-programmability The same application in JaKtA could be created by defining a *parametric* MAS via an ordinary Kotlin function with a parameter:

[8] https://github.com/jakta-bdi/jakta-examples

```
fun ticTacToe(gridSize: Int = 3) = mas {
  require(gridSize > 0);
  environment { from(GridEnvironment(gridSize)) ; actions { action(Put) } }
  player(mySymbol="x", otherSymbol="o", gridSize=gridSize)
  player(mySymbol="o", otherSymbol="x", gridSize=gridSize)
}
```

The function declares a MAS whose environment of type `GridEnvironment` of size `gridSize` supporting an external action `Put` (defined elsewhere). The two players are agents returned by the `player` extension function:

```
fun MasScope.player(mySymbol: String, otherSymbol: String, gridSize: Int) =
  agent("$mySymbol-agent") {
    beliefs {
      alignment("vertical",dx=0,dy=1); alignment("horizontal",dx=1,dy=0)
      alignment("diagonal",dx=1,dy=1); alignment("antidiagonal",dx=1,dy=-1)
      setOf("vertical", "horizontal", "diagonal", "antidiagonal")
        .forEach { rule { aligned(L) impliedBy it(L) } }
    }
    plans {
      detectVictory(mySymbol, gridSize)
      detectDefeat(mySymbol, otherSymbol, gridSize)
      makeWinningMove(mySymbol, gridSize)
      preventOtherFromWinning(mySymbol, otherSymbol, gridSize)
      randomMove(mySymbol)
    }
  }
```

Notably, the function exploits multiple paradigms to construct agent specifications via AOP meta-programming. For instance, predicate `aligned/1` is defined in a `forEach` loop, while predicates `vertical/1`, `horizontal/1`, and (`anti`)`diagonal/1` are defined by calling the `alignment` function, which parametrically builds rules to compute alignments along the four major directions:

```
fun BeliefsScope.alignment(name: String, dx: Int, dy: Int) {
  val first = cell(A, B, C); val second = cell(X, Y, Z)
  rule { name(listOf(second)) impliedBy second }
  rule { name(listFrom(first, second, last = W)) .impliedBy(
      first, second, (X - A) arithEq dx, (Y - B) arithEq dy,
      name(listFrom(second, last = W))) }
}
```

With no paradigm blending, based on the bare AgentSpeak(L) syntax, the rules would have needed to be copied and modified to support multiple cases instead.

Plans are defined by means of Kotlin functions as well: JaKtA plans can have names, meta-parameters, and leverage decomposition. For instance, victory and defeat detection are implemented with functions parametric in the symbol of the player and size of the grid:

```
fun PlansScope.detectVictory(myMark: String, size: Int) =
  detect(myMark, myMark, size) { Print("I won!") }
fun PlansScope.detectDefeat(myMark: String, otherMark: String, size: Int) =
  detect(mySymbol, otherMark, size) { Print("I lost!") }
```

and both rely on a generic `detect` function implementing a *template plan*:

```
fun PlansScope.detect(me:String,oth:String,s:Int,action:BodyScope.()->Unit) =
  +turn(me) onlyIf { aligned((1..s).map { cell(oth) }) } then(action)
```

Finally, we show how *plan generation* can be realised in JaKtA by showing the

implementation of `makeWinningMove`:

```
fun PlansScope.winningMove(myMark:String, gridSize:Int, mark:String=myMark) =
  allPermutationsOf(cell(X, Y, e), cell(mark), size - 1).forEach {
    +turn(myMark) onlyIf { aligned(it) } then { Put(X, Y, myMark) }
  }
```

There, `gridSize` plan statements are generated, one for each possible position of the empty cell in a line containing $N-1$ cells with the same mark. Once again, the definition is parametric in the size of the grid and the symbol of the current agent. In this way, the JaKtA code would work with all possible values $N > 0$, whereas the corresponding AgentSpeak(L) code would need to be tailored on a single value of N.

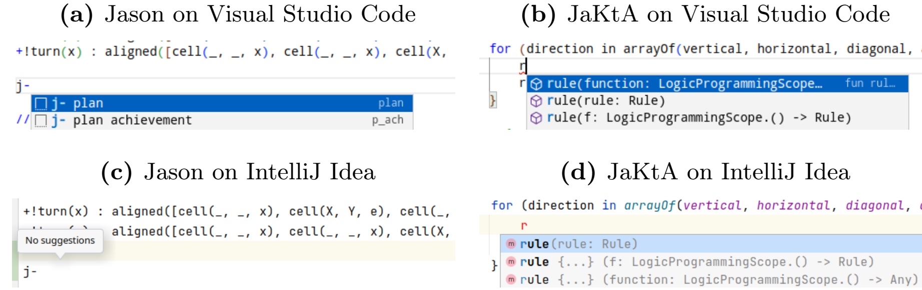
We believe that reusable units of agent behaviour such as template plans and plan generation, made possible by intertwining multiple paradigms, promote abstraction, reuse, and allow for improved code-organization.

Code organisation, reuse, and type safety Proper organisation is important to the understandability and extensibility of any program. For instance, in our example, separating the belief base from the plan library may be useful to change the latter in order to implement different strategy. The main reuse technique in Jason (similar for many other external AOP DSLs) is plain file inclusion, performed with statements of the form `include("path/to/file.asl")`. The mechanism is simple, but arguably limited and relatively unsafe, as the actual result of the inclusion will be known at runtime.

Instead, JaKtA inherits the abstraction mechanisms of Kotlin: programs can be suitably split into different pieces, at different levels of granularity (package, file, class, function). Pieces may be either individual beliefs, plans, actions, or agents, or even groups of them. Furthermore, JaKtA's (Kotlin's) reusable abstractions are *type-safe*: one cannot, for instance, include a belief where a plan is expected, and consistency is verified at compile time by the Kotlin compiler.

Tooling and ecosystem An indirect benefit of internal DSLs is the availability of inheriting the rich ecosystem of tools of the host language. We quickly exemplify in Figure 1 comparing how JaKtA and Jason are supported by two commonly used IDEs: Visual Studio Code (VSCode) and IntelliJ Idea. We install, in both cases, the latest version of the Jason and Kotlin plugins; notably, we developed nothing specific for JaKtA, so everything that is displayed came with no development and maintenance cost. As the figure shows, we get code highlighting and content assist for both languages in VSCode, although, thanks to Kotlin's type system, we obtain better completion suggestions. It is also worth noting that the suggestions for Jason are in the form of code snippets and have no real contextual relevance. On IntelliJ Idea, however, we have no highlighting or assist of any kind for Jason beyond the tools the IDE provides for plain text files: in fact, no Jason plugin for Idea exists, users coming from that IDE need to adapt to a new one, or developers need to invest time and resources into developing one. Opposedly, JaKtA is fully supported in any IDE featuring Kotlin

Fig. 1: IDE support for Jason and JaKtA compared Visual Studio Code (top) and IntelliJ Idea (bottom). By inheriting the tools made for Kotlin, JaKtA is fully supported in both IDEs with no need for additional development or maintenance.



(a) Jason on Visual Studio Code
(b) JaKtA on Visual Studio Code
(c) Jason on IntelliJ Idea
(d) JaKtA on IntelliJ Idea

support (at the time of writing, this includes VSCode, Idea, Android Studio, Eclipse, and Atom[9]).

Additionally, leveraging Kotlin as host language allows JaKtA code to be smoothly embedded in Android applications. The TicTacToe example described above has also been tested on Android[10], as demonstrated by fig. 2. JaKtA is available on Maven Central, and can thus be imported as an ordinary dependency in any Android project, at the cost of a single line in the projects' Gradle build file.

Fig. 2: The TicTacToe MAS running on Android.

5 Conclusion, limitations, and future work

In this paper, we introduce JaKtA: an internal DSL for BDI agent programming, written in Kotlin, that strives to achieve true paradigm blending of AOP, OOP, and FP in a mainstream language. We show how JaKtA can be used

[9] https://kotlinlang.org/docs/kotlin-ide.html
[10] code available at: https://github.com/jakta-bdi/jakta-android-example

to implement a simple BDI agent, and how paradigm blending can be used to achieve improved modularity, and to build reusable BDI elements, thus providing value to the authors of AOP software. Moreover, we show that, with no need for dedicated components or tools, and thus with no additional development and maintenance cost for the language developers, JaKtA is already supported by most popular IDEs, as it can rely on the existing infrastructure of its host language. Additionally, we argue that JaKtA could enable more developers to get in touch with AOP, since it does not require newcomers to learn a new language, and/or adopt new tools.

Limitations. Approaching the problem through internal DSLs provides several benefits already discussed, but they come at the expense of syntactic flexibility induced by the host language (with different languages having imposing different constraints). Thus, due to the features of Kotlin, differences among JaKtA and AgentSpeak(L) are unavoidable. Indeed, only a fixed subset of symbols can be overloaded in Kotlin. For instance, while the unary logical operator `!` can be overridden in Kotlin, the binary Elvis operator `?:` cannot. Thus, JaKtA's syntax favors explicit keywords such as `achieve` and `test` to represent achievement and test goals, respectively. Many syntactical design choices in JaKtA were driven by the need to find appropriate Kotlin representations of Jason-inspired entities. As a result, JaKtA's syntax may be more verbose than Jason's: the choice between external and internal DSLs, in general, imposes a trade-off between conciseness and reuse.

Concerning runtime behaviour, JaKtA's architecture has been designed to separate the concurrency model from the agent specification. The implementation discussed in this work relies on a sequential implementation, but different concurrency models are under active development and will be explored in a future work.

Future work. In the future, our research efforts will follow four main directions. Firstly, we plan to improve JaKtA to fully support Kotlin multiplatform facilities, thus enabling the exploitation of a single language and interpreter for running BDI systems on top embedded devices (Kotlin/Native) as well as in Web (Kotlin/JavaScript), mobile (Kotlin/Android), and general-purpose (Kotlin/JVM) applications. Second, with the help of the concurrency management module developed for JaKtA (which we plan to describe in detail in another work), we intend to investigate how different concurrency models may impact the design and performance of MASs, both in real-world and simulated scenarios. Along this line, we will also investigate how JaKtA can be integrated with mainstream simulation frameworks, to provide better support to the development of distributed MASs and we will attempt to compare how JaKtA relates to other AOP technologies in terms of performance to understand whether the possibility to change the concurrency model can achieve performance gains. Finally, we will look for ways to improve the syntax of the DSL, in order to increase its readability and to thin the gap between the OOP and AOP.

References

1. Andrei, S., Masalagiu, C.: About the collatz conjecture. Acta Informatica **35**(2), 167–179 (1998). https://doi.org/10.1007/s002360050117
2. Baiardi, Martina, Ciatto, G., Pianini, D., Semantic Release Bot: jakta-bdi/jakta: v0.3.0 (2023). https://doi.org/10.5281/zenodo.7900584
3. Bordini, R.H., Hübner, J.F., Wooldridge, M.J.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons, Ltd (Oct 2007), http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470029005.html
4. Bratman, M., et al.: Intention, plans, and practical reason, vol. 10. Harvard University Press Cambridge, MA (1987)
5. Calegari, R., Ciatto, G., Mascardi, V., Omicini, A.: Logic-based technologies for multi-agent systems: A systematic literature review. Autonomous Agents and Multi-Agent Systems **35**(1), 1:1–1:67 (2021). https://doi.org/10.1007/s10458-020-09478-3
6. Ciatto, G., Calegari, R., Omicini, A.: 2P-Kt: A logic-based ecosystem for symbolic AI. SoftwareX **16**, 100817:1–100817:7 (Dec 2021). https://doi.org/10.1016/j.softx.2021.100817
7. Ciatto, G., Calegari, R., Siboni, E., Denti, E., Omicini, A.: 2P-Kt: logic programming with objects & functions in Kotlin. In: Calegari, R., Ciatto, G., Denti, E., Omicini, A., Sartor, G. (eds.) WOA 2020 – 21th Workshop "From Objects to Agents". CEUR Workshop Proceedings, vol. 2706, pp. 219–236. Sun SITE Central Europe, RWTH Aachen University, Aachen, Germany (Oct 2020), http://ceur-ws.org/Vol-2706/paper14.pdf, 21st Workshop "From Objects to Agents" (WOA 2020), Bologna, Italy, 14–16 Sep. 2020. Proceedings
8. Cohen, P.R., Levesque, H.J.: Intention is choice with commitment. Artif. Intell. **42**(2-3), 213–261 (1990). https://doi.org/10.1016/0004-3702(90)90055-5
9. Collier, R.W., Russell, S.E., Lillis, D.: Reflecting on agent programming with AgentSpeak(L). In: PRIMA 2015: Principles and Practice of Multi-Agent Systems, pp. 351–366. Lecture Notes in Computer Science, Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-25524-8_22
10. D'Urso, F., Longo, C.F., Santoro, C.: Programming intelligent iot systems with a python-based declarative tool. CEUR Workshop Proceedings, vol. 2502, pp. 68–81. CEUR-WS.org (2019), https://ceur-ws.org/Vol-2502/paper5.pdf
11. Finin, T., Fritzson, R., McKay, D., McEntire, R.: Kqml as an agent communication language. In: Proceedings of the Third International Conference on Information and Knowledge Management. p. 456–463. CIKM '94, Association for Computing Machinery, New York, NY, USA (1994). https://doi.org/10.1145/191246.191322, https://doi.org/10.1145/191246.191322
12. Georgeff, M.P., Lansky, A.L.: Reactive reasoning and planning. In: AAAI. vol. 87, pp. 677–682 (1987)
13. Hindriks, K.V.: Programming rational agents in GOAL. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming: Languages, Tools and Applications, pp. 119–157. Springer, Boston, MA (May 2009). https://doi.org/10.1007/978-0-387-89299-3_4
14. Johann, P., Polonsky, A.: Higher-kinded data types: Syntax and semantics. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019. pp. 1–13. IEEE (2019). https://doi.org/10.1109/LICS.2019.8785657, https://doi.org/10.1109/LICS.2019.8785657

15. Körner, P., Leuschel, M., Barbosa, J., Costa, V.S., Dahl, V., Hermenegildo, M.V., Morales, J.F., Wielemaker, J., Diaz, D., Abreu, S.: Fifty years of prolog and beyond. Theory Pract. Log. Program. **22**(6), 776–858 (2022). https://doi.org/10.1017/S1471068422000102, https://doi.org/10.1017/S1471068422000102
16. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The ocaml system: Documentation and user's manual. INRIA **3**, 42
17. Mazinanian, D., Ketkar, A., Tsantalis, N., Dig, D.: Understanding the use of lambda expressions in java. Proc. ACM Program. Lang. **1**(OOPSLA), 85:1–85:31 (2017). https://doi.org/10.1145/3133909
18. Nozik, A.: Kotlin language for science and Kmath library. AIP Conference Proceedings **2163**(1) (10 2019). https://doi.org/10.1063/1.5130103, https://doi.org/10.1063/1.5130103, 040004
19. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the a&a meta-model for multi-agent systems. Auton. Agents Multi Agent Syst. **17**(3), 432–456 (2008). https://doi.org/10.1007/s10458-008-9053-x
20. Palanca, J., Rincon, J.A., Carrascosa, C., Julián, V., Terrasa, A.: A flexible agent architecture in SPADE. Lecture Notes in Computer Science, vol. 13616, pp. 320–331. Springer (2022). https://doi.org/10.1007/978-3-031-18192-4_26
21. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In: Håkansson, A., Nguyen, N.T., Hartung, R.L., Howlett, R.J., Jain, L.C. (eds.) Agent and Multi-Agent Systems: Technologies and Applications, Lecture Notes in Computer Science, vol. 5559, pp. 149–174. Springer, Berlin, Heidelberg (June 3–5, 2005). https://doi.org/10.1007/0-387-26350-0_6
22. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a bdi-architecture. In: Allen, J.F., Fikes, R., Sandewall, E. (eds.) Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91). Cambridge, MA, USA, April 22-25, 1991. pp. 473–484. Morgan Kaufmann (1991)
23. Ricci, A., Piunti, M., Viroli, M.: Environment programming in multi-agent systems – an artifact-based perspective. Autonomous Agents and Multi-Agent Systems **23**(2), 158–192 (Sep 2011). https://doi.org/10.1007/s10458-010-9140-7
24. Riti, P.: Practical Scala DSLs: Real-World Applications Using Domain Specific Languages. Apress, Berkeley, CA (2018). https://doi.org/10.1007/978-1-4842-3036-7
25. Stroustrup, B.: The C++ programming language. Addison-Wesley, 3rd edn. (1997)
26. Winikoff, M.: JACK^{TM} intelligent agents: An industrial strength platform. vol. 15, pp. 175–193. https://doi.org/10.1007/0-387-26350-0_7