SN Computer Science (2024) 5:1003
https://doi.org/10.1007/s42979-024-03244-y

ORIGINAL RESEARCH

Blending BDI Agents with Object-Oriented and Functional Programming with JaKtA

Martina Baiardi[1] · Samuele Burattini[1] · Giovanni Ciatto[1] · Danilo Pianini[1]

Received: 1 February 2024 / Accepted: 19 August 2024

Abstract

The popularity of multi-paradigm languages is on the rise, enabling developers to select the most appropriate paradigm for each task. While object-oriented and functional programming are commonly combined, other paradigms can also be hybridized. This paper introduces JaKtA, an internal Domain-Specific Language designed to support the definition of Belief-Desire-Intention agents in Kotlin. Our work represents an initial exploration into blending Agent-Oriented Programming with other prevalent paradigms, emphasizing the potential benefits of using internal Domain-Specific Languages. We demonstrate, through JaKtA, how this approach facilitates the creation of compact and expressive Belief-Desire-Intention agents that seamlessly integrate with the host language, its libraries, and tooling.

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

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 [1], which adds object-orientation on top of the functional paradigm; Java (since version 8), which supports some functional abstractions on top of its native Object-Oriented Programming (OOP) [2] via *lambda expressions* and the *stream Application Programming Interface (API)*; and Scala, that since its conception has been designed with both OOP and Functional Programming (FP) in mind [3, 4].

Additionally, some general-purpose programming languages (GPLs) feature a flexible syntax that allows to implement *internal* DSLs, capturing problem-specific abstractions but still letting the user transparently fall back to the underlying language native abstractions when needed. This is the case, for instance, of Scala, Groovy, and Kotlin: in many cases, non-core libraries are built such that using them provides a feeling (and makes the user write code code) akin to the one of a dedicated programming language. The construction of internal DSLs plays particularly well with statically and strongly typed languages, for which the Integrated Development Environment (IDE) can usually provide better contextual assistance.

Although, originally, DSLs were meant to provide dedicated abstractions for specific domains, there is no principled reason for them not to be used to model a full-fledged paradigm. Arguably, one reason discouraging such practice in most cases relates to the final ergonomics of the resulting DSL: since the DSL is built on top of the host language, it is bound to the syntactic constraints of the latter, which may end up hindering the expressiveness of the DSL too much compared to a dedicated language.

Oppositely, most AOP languages are implemented as *external* DSLs, —commonly on top of some GPL such as Java or Python—, typically featuring the possibility to call

✉ Giovanni Ciatto
giovanni.ciatto@unibo.it

Martina Baiardi
m.baiardi@unibo.it

Samuele Burattini
samuele.burattini@unibo.it

Danilo Pianini
danilo.pianini@unibo.it

[1] Department of Computer Science and Engineering (DISI), Alma Mater Studiorum, Universitá di Bologna, Via dell'Universitá 50, 47522 Cesena, FC, Italy

Published online: 30 October 2024

the underlying GPL code via a Foreign-Function Interface (FFI), a dedicated mechanism allowing calls to code written in a different language. Arguably, the interaction between an external DSL and its host language through a FFI is not as seamless as the one between an internal DSL and its host language, as GPL code is isolated from the DSL, and ad-hoc syntactical features are required to "call" it from the DSL. Furthermore, an external DSL requires users (resp. developers), to spend additional effort in learning (resp. realising and maintaining) both a new language (parser) and an ecosystem of tools (e.g., debugger, syntax coloring, linter, IDEs, etc.).

To the best of our knowledge, no mainstream programming language currently features *native* support for the Agent-Oriented Programming (AOP) paradigm, and especially for the 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) [5] semantics—such as Jason [6], Astra [7], and GOAL [8]. 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, debuggers, etc.) whose development and maintenance adds upon the cost of the language itself—potentially causing the ecosystem to evolve slowly, thus hindering adoption.

In this paper, we propose to leverage *internal* DSLs as a potential solution to both the availability in a mainstream language and the tooling support for BDI languages. This paper is an extended version of our previous work presented at EUMAS2023 [9], enriched with the feedback we received from the reviewers and the discussions we had with other experts in the field. Inspired by the successful Jason AOP language, we present Jason-like Kotlin Agents: a Kotlin internal DSL meant to seamlessly integrate BDI agents into a mainstream programming language, adding BDI-flavoured 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 boundary among the three 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 demonstrate that the tooling of the host language can be used immediately, with no need for additional support software to be developed and maintained.

Among the key benefits of the internal DSL approach, we highlight modularity, meta-programming, and interoperability. There, modularity refers to the possibility, for JaKtA developers, to exploit Kotlin's namespaces (packages, modules, classes, or methods) to partition, organise, and reuse (possibly partial) agent specifications, as they would do for ordinary Kotlin code. Meta-programming refers to the possibility to use Kotlin's OOP or FP constructs to parametrise the definition of Multi-Agent System (MAS) specifications. Finally, interoperability refers to the possibility of writing JaKtA agents that can seamlessly interact with the rest of the Kotlin ecosystem, including its standard library and toolkits—or, vice versa, to write Kotlin code that delegates portions of its operations to JaKtA agents.

Accordingly, in the remainder of this paper, we discuss the design rationale 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 order to assess the effectiveness of our internal DSL approach, we compare JaKtA to state-of-the-art BDI agent programming technologies —most notably, Jason—, and we exemplify the usage of JaKtA in a few case studies.

The remainder of this paper is organised as follows: in "Background", we present DSL engineering and we summarise the state of the art of BDI languages, then in "A Kotlin DSL for BDI Agents" 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 "JaKtA in Practice: Running Example" 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 "Conclusion", we conclude the paper by discussing some limitations of our approach, as well as some future research directions stemming from it.

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

DSL engineering

As introduced in "Introduction", 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 AgentSpeak(L) 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 [3]: *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" [10]), 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, Groovy) provide syntactic features specifically tailored to the constructions of internal DSLs. Although these features make internal DSLs viable, they can hardly provide an expressiveness comparable to that of an external DSL, as they are still bound to the host language syntax.

A notable example of a feature designed for DSL engineering which is at the same time an enabler and a limit is the lambda expression syntax of Kotlin and Groovy. In both languages, the following evaluates to an anonymous function: `{ arguments -> body }`, which can be further simplified to `{ body }` for 0-ary lambdas or 1-ary lambdas using `it` as implicit name for the single argument. This peculiar syntax is designed to be used effectively in conjunction with the *trailing lambda convention*, by which any function accepting a lambda expression as its last argument can be invoked by moving the lambda expression outside the round parentheses; additionally, if the lambda expression is the only argument, the round parentheses can be omitted altogether. For instance, a function with signature `fun myBlock(body: () -> Any)` taking a single 0-ary lambda expression as its argument can be invoked as `myBlock{"I love DSLs"}`, providing the feeling of a custom block of code named as the higher-order function `myBlock`. Although this feature makes the construction of new DSL blocks particularly easy, the price to pay is that most DSL constructs must be enclosed in curly braces, thus limiting the DSL design freedom.

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.

In fact, besides the language ergonomics itself, the success of a language is also determined by the availability of a rich ecosystem of libraries and tools (syntax highlighting, code completion, live error detection, assisted refactoring, debuggers, linters, profilers, etc.), lowering the learning curve and supporting advanced practical uses. Here lies a key motivation for favoring internal DSLs over external ones: most (often, all) of the pre-existing tooling for the host language can be reused for the new internal DSL, reducing *both* maintenance costs and learning curve. From the point of view of developers and maintainers, time and resources that would have been spent into developing and maintaining part of the tooling and documentation can be redirected to other improvements; from the point of view of users, pre-existing knowledge of the host GPL language and its tooling can be leveraged to quickly start with the new DSL, and, for the newcomers, most of the documentation and tutorials for the host language can be reused.

Kotlin as a Host Language for Internal DSLs

Internal DSL-like solutions may be realised in virtually any GPL, for instance by defining a *fluent* API [3]. In that case, users would write their DSL code by chaining method calls, while still using ordinary OOP constructs. However, recent languages such as Scala, Kotlin, or Ruby come with a flexible syntax which gives DSL designers great flexibility in extending the hosting GPL with new constructs.

In the particular case of Kotlin as the hosting GPL, internal DSL design leverages primarily the following set of features[2] [11]:

Operator overloading[3]—allowing ordinary arithmetic, comparison, access, assignment, and function invocation operators to change their ordinary meaning on a per-type basis;
Block-like lambda expressions and trailing lambda convention[4]—see "DSL engineering"
Extension methods[5]—allowing pre-existing types to be extended with new instance methods whose visibility is scope-sensible.

[1] Martin Fowler's blog post on DSL Boundary archived at https://archive.is/xAeiX.

[2] A comprehensive review of Kotlin's features that can be exploited for building DSLs is beyond the scope of this paper, but we refer the interested reader to the Kotlin documentation about the so-called "type-safe builders", archived at https://archive.is/El3fE.

[3] Kotlin Operator overloading documentation, archived at https://archive.is/m36Pl.

[4] Kotlin Higher-order functions and lambdas documentation, archived at https://archive.is/ym8Gf.

[5] Kotlin Extensions documentation, archived at https://archive.is/G9DDM.

Function types/literals with receiver[4]—allowing functions and methods to accept lambda expressions within which the `this` variable references a different object than the outer scope;

Crucially for our purpose, Kotlin-based DSLs inherit all the host language's features and libraries.

Multi-agent Systems, BDI, and Their Technologies

The philosopher Michael Bratman described humans' practical reasoning via the "beliefs, desires, intentions" framework, as a way to explain future-directed decision-making [12]. Successively, the framework was formalised by means of modal logics [13], and then turned into an abstract semantic for computational agents: AgentSpeak(L) [5].

Computational agents are *autonomous* entities [14] situated into an *environment* they can perceive and affect; they interact either directly or through the environment [15] by means of stigmergy. The classical implementation of BDI agents, based on the Procedural Reasoning System (PRS) [16], is characterized by four main abstractions, namely:

Beliefs: a set of facts and rules representing an agent's *epistemic* memory, possibly describing its knowledge about the world, itself, and other agents;
Desires: a set of goals, representing (possibly partial) descriptions of desirable states of the world the agent is willing to achieve, test, or maintain;
Intentions: a set of tasks the agent is currently committed to, in order to satisfy some of its desires;
Plans: a set of *recipes* representing the agent's procedural memory, hence encoding the know-how about achieving a given intention under certain conditions.

Any instance of the aforementioned abstractions in a BDI agent may vary during its lifetime. For instance, novel beliefs appear in the agents' minds whenever they receive novel perceptions from their sensors, while stale beliefs simultaneously disappear. Similarly, novel beliefs may arise while agents interact with each other—or with humans—or as they choose to memorize some information they have deducted via reasoning. The occurrence of relevant events may induce an update in the set of desires (i.e., acquiring new goals to be achieved/tested/maintained and/or discarding some goals). Agents' desires eventually lead to dropping existing or adopting novel intentions (activities to achieve/test/maintain goals the agent is committed to).

While carrying on an intention, agents may select one or more plans among those supporting the corresponding intention's accomplishment. Plans may involve the execution of one or more actions – possibly affecting the environment via actuators – or the accomplishment of further sub-goals, which may, in turn, require the execution of further plans as part of the same intention. Finally, agents could even learn entirely new plans, either through reasoning or through information sharing with other agents.

Interaction

Widening the scope to *multi*-agent systems, in which BDI agents are able to exchange information among each other, the focus shifts from the single agent to the interplay among multiple agents. Communication can happen directly through explicit agent-to-agent communication or indirectly through a change in the environment. The latter is also called stigmergic communication [17, 18]: the actions of one agent may alter the environment, and this alteration may be perceived by other agents, which in turn adapt their behaviour. To support this capability in software systems, MAS technologies come with explicit and malleable environment abstractions [19] which MAS developers can customise to accommodate the application's needs. In practice, the environment contains both agents *and* information that agents can access and modify; thus, specific abstractions are provided to perform such operations, typically ad-hoc actions or artifacts [20].

Communication

Communication is a more explicit form of interaction where agents exchange messages among each others. Message exchanges may, in turn, be structured in conversations, matching well-recognised patterns, also known as interaction protocols.[6]

In interaction protocols, each interacting agent plays a specific role, and it is supposed to send messages only in specific moments, to specific agents, and with specific contents.

At the technical level, MAS technologies must provide some low-level message-passing mechanism between agents. Solutions in this case range from plain method calls – limiting interactions to agents sharing the same process – to distributed solutions, possibly involving network protocols and infrastructural components—such as message queues, brokers, or other middlewares, which let agents interact across different machines.

At the semantic level, interaction protocols require Agent Communication Languages (ACLs). ACLs are high-level message formats, purposefully designed to include meta-data that lets agents interpret the meaning of messages [21]. One notable example is the Knowledge Query and Manipulation Language (KQML) [22], which has a dual nature of message format and ACL, with a focus on how knowledge is shared among agents. Among the many meta-data prescribed by KQML, the message *performative*, defining the *purpose* of a message with respect to the protocol and the sender's role, is likely the most relevant one. Admissible KQML performatives derived from speech-act theory [23]

[6] FIPA Interaction Protocols, archived at https://archive.is/phy6v.

are many, thus modern BDI technologies such as Jason commonly rely on a subset of – there referred to as "illocutionary forces" [6] –, which cover scenarios such as transferring beliefs (`tell`), transferring achievement (`achieve`) or test (`askOne`, `askAll`) goals, or transferring plans (`askHow`, `tellHow`).

BDI Technologies Comparison

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. For the sake of completeness, we included the current status of JaKtA in the comparison, even though its realisation is successive to the analysis of the other languages that we discuss in this section. Details on the choices we made for JaKtA are discussed in "A Kotlin DSL for BDI Agents".

As far as BDI agent programming languages are concerned, our comparison focuses on those languages that appear to have some running software implementation that is actively maintained and used by the community. Hence, we build upon the recent work by Calegari et al. [24], which surveys the state-of-the-art of logic-based agent-oriented technologies, and we select the ones aimed to support general-purpose BDI agents programming. We remark that this comparison is not meant to be exhaustive, as providing a complete overview of all the BDI agent programming languages available in the literature is beyond the scope of this paper; rather, we show a panorama of the technologies we have been inspired by when designing JaKtA. In this sense, our comparison aims to identify both desirable features and potential limitations of the sampled technologies, in order to take them into account when designing a novel BDI technology.

As far as features are concerned, in the remainder of this section we discuss the ones we believe are most relevant:

- full **AgentSpeak(L) compliance**;
- **DSL type** (internal or external);
- **hosting syntax**, i.e., which syntax the DSL is embedded in (for internal DSLs) or based upon (for external DSLs);
- **execution platform**, i.e., which runtime platform the language runs upon;
- **interoperability**, i.e., whether and how other languages can be called from within the BDI language (and, in that case, which ones);
- **paradigm blending**, i.e., whether it is possible to mix, in the same source and scope, AOP and other abstractions;
- **type safety**, i.e., the ability of the compiler/interpreter to intercept (most) type errors ahead-of-execution;
- **reuse mechanisms**, i.e., whether and how it is possible to parameterise and reuse partial or entire MAS specifications;
- **concurrency model** i.e., whether the MAS platform supports developers in switching between multiple ways to execute agents;
- **explicit environment**, i.e., whether the language provides it as an explicit *and* programmable abstraction;
- **internal actions**, namely, whether the language provides mechanisms to implement agent introspection (i.e., observing the agent's own state) and related reasoning;
- **external actions**, akin to internal actions, but related to the perception of and actuation over the environment;
- **message-passing mechanism**, i.e., how intercommunication among agents is actually supported; and, finally,
- **license**, which can impact severely on the adoption of a technology.

Full AgentSpeak(L) compliance

We distinguish between fully compliant BDI technologies and the ones that are just inspired to AgentSpeak(L). Since such compliance, to the best of our knowledge, has not yet been formally defined, for the scope of this paper we rely on the following set of necessary features:

1. explicit representation of beliefs, goals, plans, and events in the language syntax;
2. explicit representation of belief bases, events, plan libraries, and intentions in the engine;
3. all the aforementioned representations are based on logic terms and clauses, thus including:
   (a) arbitrarily structured, possibly nested and non-ground, logic terms and clauses;
   (b) logic unification and resolution (cf. [29]) to manipulate the aforementioned data structures;
4. event-based architecture for the execution of the reasoning cycle;
5. explicit support for (at least) achievement and test goals.

Our analysis identified that both Jason and SPADE-BDI are fully compliant to the AgentSpeak(L) definition, in fact, they both can describe agents by the means of `.asl` files, which are structured to represent beliefs, goals, plans, and events following the abstract semantics of AgentSpeak(L). Furthermore, PHIDIAS adheres to our definition of AgentSpeak(L) compliance as well, even though it does not support the AgentSpeak(L) syntax directly, as they come with an internal domain-specific language. GOAL supports logic representation, but is missing intentions and test goals, hence we

Table 1 Comparison of the identified practical features across several common BDI agent programming languages

	JaKtA	Jason [6]	SPADE-BDI [25]	PHIDIAS [26]	Astra [7]	JACK [27]	Jadex [28]	GOAL [8]
AgentSpeak(L) Compliance	✓	✓	✓	✓	×	×	×	×
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
Interoperability	Native Java, Kotlin FFI native	FFI Java	Native Python FFI native	Native Python FFI native	FFI Java	FFI Java	FFI Java	Native SWI-Prolog
Paradigm blending	✓	×	✓	✓	✓	✓	×	✓
Type safety	✓	×	×	×	✓	✓	✓	×
Reuse mechanisms	Any Kotlin mechanism	file incl.,internal actions	Any Python mechanism	Any Python mechanism	agent extension	reusable plans and capabilities	selective file incl. Java inheritance	reusable plans, beliefs, goals, and agents
Concurrency Model conf.	✓	✓	×	×	✓	~	✓	✓
Explicit Environment	✓	✓	×	×	✓	×	×	✓
Internal Actions	✓	✓	✓	✓	✓	×	×	✓
External Actions	✓	✓	×	×	✓	×	×	✓
Message Passing	Pluggable	Pluggable	XMPP	HTTP TCP Sockets	UDP	Pluggable	TCP/SSL	In process
License	Apache 2.0	LGPL v3	GPL v3	MIT	GPL v3	Proprietary	GPL v3	GPL v3

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

classify it as not fully compliant. Similarly, even though Astra supports all the AgentSpeak(L) features, it does not fully support logic programming to represent beliefs, goals, plans, and events. Jadex does not provide any unification mechanism. Finally, although JACK explicitly models beliefs and plans, it does not represent them using logic terms; rather, its *beliefs sets* work as in-memory relational databases. Thus, we classify JACK as not AgentSpeak(L)-compliant.

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 understanding which syntax the DSL is embedded in, whereas, for external DSLs, we further describe the derivation of the syntax. Accordingly, regarding internal DSLs, both SPADE-BDI and PHIDIAS are hosted by Python. Conversely, the syntax of external DSLs is built as an extension or refinement of a GPL. For instance, while Jadex relies on XML, GOAL extends Prolog [29], 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 Java Virtual Machine (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. Considering the languages under comparison, SPADE-BDI and PHIDIAS target Python, while all the other ones target the JVM.

Interoperability This feature concerns the ability of the agent programming language to interact with the host language constructs. Interaction can be achieved in three different ways:

1. *native*: the AOP language is capable of directly calling the GPL language constructs;
2. *FFI*: the AOP language features a dedicated API allowing streamlined interaction;
3. *indirect*: communication is possible, but it requires manually concocted workarounds (e.g., communication via sockets or files).

We consider in this discussion only the two former types of interoperability. Internal DSLs such as PHIDIAS and SPADE-BDI are capable of native interoperability by constructions (they are valid fragments in the host GPL), while external DSLs require, typically, the use of a FFI. All the languages targeting the JVM we considered can interoperate with Java constructs, but in many cases this interaction must be carefully crafted, adapting to an existing API, which may require the creation of glue code. For instance, in Jason the programmer must adhere to pre-defined Java interfaces provided by the framework, producing pure Java code that can then get called from Jason. Once Java interoperability is established, it is often possible to call also other JVM languages (Scala, Kotlin, Groovy, etc.), limited to the fragments that produce bytecode overlapping Java's.[7] However, notice that the presence of a FFI does not tell the entire story about *ease* of interoperability. For instance, although implemented in Java, Jadex requires the MAS specification to be written in XML, making interoperability with Java code cumbersome compared to other JVM-based AOP languages.

Paradigm blending We call paradigm blending the *syntactical* language capability to mix, in the same code fragment, AOP constructs and GPL-native constructs implementing a different paradigm's abstractions, such as object creation and manipulation (for OOP languages) or higher-order functions definition and invocation (for FP languages).

Notice that whether blending is beneficial or detrimental depends on the goal of the language. As a rule of thumb, blending tends to be beneficial when building complex applications, of which different parts may be better expressed in different paradigms. On the other hand, it may be a double-edged sword when *learning* a new paradigm, as the inexperienced may be tempted to work around problems more easily tackled with AOP using other paradigms they feel more comfortable with. Notice, in fact, that some languages purposely enforce a clear separation among high-level AOP constructs (e.g., belief, goals, plans) and the hosting language ones (e.g. classes, functions, etc.), purposely allowing interoperation only through carefully-crafted FFIs [30]. The strong separation is typical of Jason and Jadex. There, in the

[7] Producing fully-Java-interoperable bytecode from languages targeting the JVM other than Java is often possible, but it requires special care. For instance, Kotlin recommends using annotations to explicitly define how Kotlin constructs should be translated into Java ones, see the Kotlin's documentation on "Calling Kotlin from Java" archived at https://archive.is/HqEIG. Scala features many mechanisms that are not directly translatable into Java, such as implicits, higher-kinded types, mixins, and multiple argument lists. The official compatibility guide provides only general information on cross-compatibility (see the archived documentation about "Interacting with Java" at https://archive.is/KX49E); however, compatibility guides are available from third parties such as Databricks, which recommends not using several Scala features in order to make the code compatible (see the "Java interoperability" section of the guide archived at https://archive.ph/Qc9Gd). For even more conceptually distant languages such as Clojure, the operation may be borderline impossible: according to the official documentation archived at https://archive.is/AeNcE, Interoperability requires the use of a dedicated Clojure FFI library. The take-away message is that native or FFI-based interoperability with Java does not automatically imply interoperability with other JVM languages.

former case, MASs are composed by scripts describing agent specifications supporting solely AgentSpeak(L)-compliant constructs, and, in the latter, by actions/environment written in ordinary Java.

Interestingly, languages whose syntax is blended with the hosting language may be further categorized into two subsorts, depending on whether they (i) limit the use of the GPL's abstractions into agents/plans specifications, or (ii) allow the GPL's constructs to appear into AOP programs directly.

Case (i), for instance, characterizes Python-based BDI languages such as SPADE-BDI and PHIDIAS, where AOP specifications consist of Python classes and methods. Conversely, Astra, JACK, and GOAL adopt strategy (ii), as their syntaxes permit writing Java or Prolog constructs and libraries, respectively.

Type safety This feature refers to the presence of a strict compile-time type checker for the BDI language at hand, which may proof check agents specifications before execution.

Solutions featuring tight interoperability with Java, such as Astra, JACK, and Jadex, come with this feature; whereas the other languages in this comparison do not. Those (Jason, SPADE-BDI, PHIDIAS, and GOAL) favour a flexible syntax over type safety, 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 inclusion mechanisms. This is the case, for instance, of Jason, supporting the inclusion of ASL files into other ASL files; and Jadex, which, in addition to Java inheritance between agent classes, supports referencing XML or Java files into other XML files for more flexible reuse of agents' capabilities.

Furthermore, virtually all surveyed solutions support the abstraction and reuse mechanisms of the hosting language, if any. For instance, internal actions in Jason are defined as pure Java classes, thus they can be reused across multiple projects. This implies, for instance, that solutions based on a OOP host GPL may take advantage of its OOP abstraction mechanisms, such as sub-typing and inheritance, for the OOP-written 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 provide ad-hoc syntax 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 reused along the MAS specification. For example, Astra supports writing agents' specifications extending other agents' specifications, whereas JACK bases its modularity on the concept of capabilities [31] wrapping beliefs and plans.

Concurrency model

This feature is about how the surveyed solutions deal with agents' *concurrent execution*, and whether developers can select among multiple execution models without resorting to changes in the execution platform's code. Despite agents having their own *logical* control flow—as required by their computational autonomy—, how such control flow is mapped in practice over processes, threads, and other platform- and operating system-specific concurrency facilities may vary wildly. The mapping between logical and physical control flows may affect several runtime properties of a MAS: performance and scalability are the most obvious one, but also determinism and reproducibility (hence, testability) are deeply affected.

Admissible choices in this case are manifold. The most common ones are:

One-Agent-One-Thread (1A1T) where each agent is mapped over a single thread;
All-Agents-One-Thread (AA1T) where all agents in the system share the same thread;
All-Agents-One-Executor (AA1E) where all agents in the system share the same executor service;
One-Agent-One-Process (1A1P) where each agent is mapped over a single process.

Jason, Astra, Jadex and GOAL provide mechanisms that allow developers to customise execution strategies for their MAS. They provide support for multiple strategies out of the box (of which one, AA1E is set as default), and they provide APIs to regulate the amount of threads in the executor.

PHIDIAS, instead, is hard-coded to 1A1T. SPADE-BDI, relying on Python coroutines that are single-threaded by default,[8], implicitly adopt the AA1T model. In the case of JACK, the documentation does not discuss the available concurrency abstractions, nor it explains how to select them or provide a custom one.

Explicit environment As introduced in "Multi-agent Systems, BDI, and Their Technologies" the design of environments is essential for multi-agent systems that rely on stigmergy [19]. In this paragraph, we analyse whether the BDI languages under scrutiny feature an explicit and *programmable* notion of environment. To the best of our knowledge, the environment is modelled as a first-class programmable construct in Jason, Astra, GOAL.

Internal/External actions support This feature is about whether BDI languages come with some means to enrich agents' or environments' specifications with custom functionalities written via the host language. Using Jason's nomenclature, we call these functionalities "external (resp.

[8] Developing with `asyncio` Concurrency and Multithreading, Python 3 docs. Archived at https://archive.is/YWQw5.

internal) actions", if they extend the agents' capability to perceive/affect the environment (resp. their own internal state). From a programming language perspective, actions are the technical bridge between the agent- and the object-oriented portions of a MAS specification.

The support for internal or external actions is heterogeneous among BDI languages (cf. Table 1), and we identified that Jason, Astra, and GOAL support both of them. For those languages for which we did not identify an explicit notion of environment, we assumed that the mechanism of external actions is not supported as well. Neither JACK nor Jadex provide first-level abstractions for actions, instead, both provide customisation mechanisms, the first one through the concept of "capability" [31], while the second leveraging Java inheritance.

Message passing mechanism Despite most BDI languages coming with primitives enabling agent interaction via message-passing, they may differ in the way they support message exchange in practice. This feature is about *how* BDI languages support message exchange. Options involve: (i) in-process communication, enabling communication only among agents sharing the same runtime; (ii) distributed communication, possibly involving data exchanges over networked machines; and (iii) customisable communication, in which, both the previous cases can be supported, depending on the configuration.

Case (i) is native in GOAL [8]. Conversely, SPADE-BDI, Astra, PHIDIAS and Jadex are characterized by case (ii), as they rely upon the XMPP, UDP, TCP sockets or HTTP, and TCP/SSL protocols respectively. Jason and JACK [27] fall into Case (iii).

License This feature is about which license BDI solutions are distributed with, which is crucial for the adoption of the technology. Mainly, solutions can require a fee for their usage, being free of charge but not open source, or being open source (hence, inspectable and extendable). All solutions but JACK (which is proprietary, commercial, and closed-source) adopt a Free and Open-Source Software (FOSS) license.

A Kotlin DSL for BDI Agents

From "Multi-agent Systems, BDI, and Their Technologies", we identify the core features that we need into an implementation of a DSL for BDI agents familiar to BDI experts and idiomatic to the community of mainstream developers. Thus, as first step, we want it to be AgentSpeak(L)-compliant and, at the same time, based on a mainstream language, hence, an *internal* DSL.

Then, the language should be statically and strongly typed, possibly featuring type inference to reduce ceremonial code. Among the other features, we considered the target platform and its portability across multiple platforms (as we wanted to maximise the range of potential target runtimes), the existing ecosystem (as we wanted 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), and 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 richer type system (supporting, for instance, path-dependent and higher-kinded types [32]) for merely practical reasons:

1. 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;
2. we expect Kotlin's popularity to grow faster than Scala's in the future, as Google picked Kotlin as reference language for the Android ecosystem,[9] and
3. Kotlin has better support for multi-targeting (JVM, JavaScript, and native code including Apple-specific platforms).

The framework has been released under a permissive FOSS license. It is available on GitHub[10] and Maven Central;[11] to ensure future references and reproducibility, every new release is also archived on Zenodo [33].

Overview of JaKtA's syntax

In this subsection we provide an overview of JaKtA from a syntactical perspective. Details about JaKtA's architecture and implementation giving semantics to the syntax are left to "Under the Hood of JaKtA".

The JaKtA's DSL syntax is strongly inspired by Jason and it is AgentSpeak(L)-compliant. We will not cover all details of the syntax and semantics of AgentSpeak(L) here, the interested reader may refer to [6].

Program entrypoint

The entrypoint of a MAS specification is the `mas` block, inside whose scope all the elements composing a BDI MAS can be defined:

```
mas {
  environment { /* ... */ }
  agent("id1") { /* ... */  }
```

[9] Android's Kotlin-first approach, archived at https://archive.is/C8aR8.

[10] https://github.com/jakta-bdi/jakta.

[11] https://search.maven.org/artifact/it.unibo.jakta/jakta-dsl.

```
  agent("id2") { /* ... */ }
}.start()
```

Commonly, users configure the environment, and provide initial agents' specifications within this block. Intuitively, The environment is configured in the `environment` block, and it may impact all agents in the MAS. Single agent specifications are provided by means of the `agent` block, specifying the agent's name.

From this first snippet, it is evident how the Kotlin syntactic features are being leveraged to build the DSL. More specifically, both `mas` and `environment` are a 1-ary functions taking, as a single parameter, a function (with receiver), which in this example is provided as a lambda expression (notice the curly braces syntax); since it is the sole parameter, the round braces of function invocation can be omitted altogether as per the trailing lambda convention. Similarly, `agent` is a binary function function taking a `String` as first parameter and a function (with receiver) as the second parameter. In this case, the round braces of function invocation cannot be omitted, yet the lambda expression can be passed outside them (again, trailing lambda convention), providing the feeling of a custom keyword and related code block. Through this kind of mechanism, JaKtA injects AOP in the form of perfectly legitimate Kotlin code.

Finally, the snippet shows how the MAS can be directly executed, by invoking the `start` method. As JaKtA is an internal DSL, the whole specification is done through valid Kotlin code. We can though distinguish two phases: (i) first, the Kotlin code within the `mas` block is executed assembling the structure of the MAS from its declarative definition (ii) then, when the `start` method is invoked, the MAS is executed by the BDI interpreter described in "Under the Hood of JaKtA".

Agents Agents are named entities created with the `agent` function. Inside the `agent` block (the lambda expression passed as last parameter) `beliefs`, `goals`, internal `actions` and `plans`, can be defined in homonym blocks.

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 athletes' names from a public web page indexing Olympic athletes, we extract ten names through a regular expression, and then we create one agent for each athlete:

```
val athletesWebPage = URL("https://
    olympics.com/en/athletes/")
val athleteNamePattern = Regex("""<
    span class="athlete--name no-trunc
    ">(.+?)<\/span>""")

mas {
  athletesWebPage.readText() // reads
      the Web page as HTML text
    .let { athleteNamePattern.findAll(
       it) } // selects atheles'
       names spans
    .take(10) // takes the first 10
       athletes
    .map { it.groupValues[1] } //
       selects atheles' names
    .map { it.replace(" ", "-") } //
       replaces spaces with dashes in
        names
    .forEach {
      agent("athlete-$it") { //
         creates one agent for each
         athlete
        beliefs { /* ... */ }
        goals { /* ... */ }
        plans { /* ... */ }
      }
    }
}
```

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 web URLs to athletes.

Beliefs

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 an API) the logic programming toolkit for Kotlin 2P- KT [34] and its internal DSL for Prolog [11].

For instance, in the following, we define a belief base containing information about paths in a graph, by means of logic facts, as well as logic rules for computing whether some location `X` is reachable from another location `Y`:

```
agent("moon walker") {
    beliefs {
        fact { "path"("location1", "
            location2") }
        fact { "path"("location2", "
            location3") }
        rule { "reachable"(X, Y)
            impliedBy "path"(X, Y) }
        rule { "reachable"(X, Z)
            impliedBy "path"(X, Y) and
             "reachable"(Y, Z) }
    }
}
```

Under the assumption that the graph represents some sort of map from the real world, the above belief base can be exploited by the agent to reason about reachability among any two locations in the map; in fact, through 2P- KT, JaKtA fully supports Prolog's unification and resolution mechanisms.

Goals

Goals can indicate either something that the agent wants to `achieve` by finding an appropriate plan, or something that it wants to `test` (discover), prioritising the consultation of the knowledge base over the execution of plans.

Similarly to Jason, JaKtA supports the definition of agents' *initial* goals (further goals may arise during the execution of the MAS) by means of the `goals` block:

```
agent("name here") {
    goals {
        achieve("goal"(X))
        test("goal"(Y))
    }
}
```

Plans

Plans describe which operations the agent is capable to perform to reach its goals. Inheriting the successful model of Jason, in JaKtA, plans are composed of a **triggering event** deciding whether the plan is *relevant*, an optional **context** restricting its *applicability*, and a **body** listing the operations to be performed whenever the plan is executed.

The triggering event can be a goal/belief invocation/addition (+) or failure/deletion (−), in the form: [+|−] < `triggering event>` `onlyIf` {<context>} `then` {<body>}. We inherit from Jason the usage of a prefix unary + to indicate invocation or additions, and a prefix unary − to indicate failures or deletions.

If a logical expression is present in the context block (prefixed by `onlyIf`), it is then used to vet the relevant plan. The condition therein expressed should be a logic formula to be tested against the belief base via logic resolution.

Finally, if the plan is selected for execution, the sequence of operations and actions contained in its body (prefixed by `then`) is performed by the agent. There, actions may consist of edits (additions or deletions) to the belief base, as well as additions of further achievement or test goals, or invocations of external or internal actions (see next paragraphs).

```
plans {
  +/-achieve("goal"(X)) onlyIf { "
      guard"(X) } then {
       achieve("goal"(X))
       test("goal"(Y))
       spawn("goal"(Z))
       +/-"belief"(A)
       update("belief"(C))
       execute("action"(X, Y, Z))
  }
  +/-test("goal"(Y)) onlyIf { "guard"(
      Y) }
       then { /*...*/ }
  +/-"belief"(Z) onlyIf { "guard"(Z) }
       then { /*...*/ }
}
```

Internal Actions Actions are (usually imperative) operations that agents may invoke within plans. They are meant to be used when some computation can not be encoded conveniently in a declarative way. They may also be used to get "under the hood" and inspect or modify the MAS internals.

In particular, *internal* actions come with an ad-hoc APIs for inspecting or modifying the agent's state, whereas external actions provide APIs for inspecting or modifying the environment.

JaKtA provides an ad-hoc syntax for defining and invoking internal actions.

Definitions occur within an agent's `actions` block, requiring name, arity, and body (the latter, as usual, in form of a lambda expression). Inside the body, the programmer can call Kotlin APIs dedicated to access and modify the agent's state, and to manipulate BDI data structures (beliefs, goals, plans, intentions).

Actions execution may occur within plans' bodies, by means of the `execute` function. This function takes as input the name of the action to be executed and (if any) the actual arguments to be passed. The same syntax is used to invoke *external* actions.

The main difference between internal and external actions is that the former are agent-specific, while the latter are shared. In practice, this means that each agent may only invoke internal actions defined within its `actions` block, whereas it may invoke any external action defined in the environment.

In the following snippet, we exemplify the general syntax for defining internal actions, and the operations therein supported for agents' state manipulation. Since this snippet is meant to be a general example, we the syntax ⟨⟩ to enclose placeholders for actual code. We will use such syntax also in other snippets that follow this one.

```
agent(⟨Agent name⟩) {
  actions { // internal
    action(⟨Action name⟩, ⟨Arity⟩) {
      addBelief(⟨Belief⟩)          ;
          removeBelief(⟨Belief⟩)
      addIntention(⟨Intention⟩)    ;
          removeIntention(⟨Intention⟩)
      addEvent(⟨Event⟩)            ;
          removeEvent(⟨Event⟩)
      addPlan(⟨Plan⟩)              ;
          removePlan(⟨Plan⟩)
      stopAgent()
      sleepAgent(⟨Time in milliseconds⟩)
      pauseAgent()
    }
  }
}
```

More precisely, internal actions provide primitives that can change the overall agent's state, not limited to its belief base, by: (i) add/remove intentions, (ii) add/remove events, that will become new agent's goals, (iii) add/remove plans, and (iv) pause or stop the agent.

Conversely, in the following snippet we exemplify the general syntax for invoking actions:

```
agent⟨Agent name⟩ {
  actions { action(⟨Action name⟩, ⟨Arity⟩) {
      /* ... */ } }
  plans {
    +achieve("goal"(X)) then {
      execute(⟨Name⟩, arg_1, ..., arg_
          ⟨Arity⟩)
    }
  }
}
```

Internal actions in JaKtA are used frequently, thus, it is common for developers to share them across multiple agents, or even across agents of different projects, building libraries of actions. Although Kotlin provides multiple means that favor reuse, the way we recommend as JaKtA developers is to create a Kotlin `object` (a singleton object under the OOP abstraction) implementing the `InternalAction` `interface` and possibly extending the `AbstractInternalAction` `abstract class`. Such `object` would provide a reusable internal action, which, as any other Kotlin `object`, can be shared across agents, shipped within libraries, and imported in other projects. For instance, the `Print` internal action, shown below, lets agents log on the console:

```
package jakta.examples

object Print : AbstractInternalAction(
    "print", 2) {
  override fun action(request:
      InternalRequest) {
    println("[" + request.agent.name +
        "] " + request.arguments.
        joinToString(" "))
  }
}
```

Internal actions defined in this way can be imported in one agent's `actions` block, similarly to actions defined on the fly, and invoked by means of the corresponding Kotlin object's name:

```
import jakta.examples.Print

agent("name"){
  actions { action(Print) }
  plans {
    +achieve("greet") then { Print("
        Hello World!") }
  }
}
```

To support the most common use cases, JaKtA comes with several ready-to-use internal actions out of the box. `Print`, exemplified above, is one of them, additional ones include `Fail`, forcibly failing the current plan; `Stop`, forcibly terminating the current agent's lifecycle; `Pause`, pausing the execution of the agent indefinitely; and `Sleep`, pausing the execution of the agent for an amount of milliseconds.

Environment

Environments are first-class citizens in JaKtA. Each MAS must have one environment, which can be configured once for the whole MAS, by means of the `environment` function. Technically speaking, a MAS environment is an instance of a class implementing the `Environment` interface. All agents in the MAS have access to the environment, which decides what agents can perceive, how they may act, and how agent-to-agent messages are delivered. Environments may also store custom data (in the form of key-value pairs), which may be made visible to (or modifiable by) agents by means of ad-hoc external actions. In the `environment` configuration block, users may define or reference the *external* actions that *all* agents may use, or reference a custom implementation of the `Environment` interface.

The following snippet exemplifies the environment configuration.

```
mas {
  environment {
    from(CustomEnvironment(⟨Args⟩)) //
        Environment class implemented
        elsewhere
    actions { /* external actions here
        */ }
    data["my.key"] = "custom value" //
        custom data store for the
        environment
    perception {
      percept { "my_key"(data["my.key"
          ]) } // all agents will
          perceive this
    }
  }
}
```

Custom implementations of `Environment` may be used if passed to the `from` function. To simplify the construction of custom environments, we provide in the JaKtA API an `AbstractEnvironment` `abstract class`; also in "JaKtA in practice: running example" we provide examples of custom environments. When omitted, the system falls back to the default implementation, working well in the majority of the cases. The `actions` function, in this context, is used to define or reference external actions. The `data` property is used to define custom data that may be accessed by agents (either for reading or writing) by means of perception or external actions. Finally, the `perception` block may be used to define which percepts shall pop up in the agent's belief bases at runtime. Of course, these percepts may read any information stored into the aforementioned `data` property.

External actions External actions are the *only* way for the agents to affect the environment or to communicate with other agents. Their design is akin to internal actions, despite the fact that they come with different capabilities, and therefore a different API that can be exploited to (i) add/remove

agents, (ii) send/broadcast messages, or (iii) add/remove/update data in the environment. as shown in the following snippet:

```
environment {
  actions {
    action(⟨Action Name⟩, ⟨Arity⟩) {
      addAgent(⟨Agent⟩)
                                ;
          removeAgent(⟨Agent id⟩)
      addData(⟨Key⟩, ⟨Value⟩)                ;
          removeData(⟨Key⟩)
      updateData(⟨Key⟩, ⟨Value⟩)
      sendMessage(⟨Agent id⟩, ⟨Payload⟩)    ;
          broadcastMessage(⟨Payload⟩)
    }
  }
}
```

As for internal actions, external actions can be defined in Kotlin by implementing the `ExternalAction` interface, typically using `object`s, and possibly extending the `AbstractExternalAction` `abstract class`. For instance, the `Send` external action, shown below, sends messages to other agents, given their name:

```
object Send : AbstractExternalAction("
    send", 3) {
  override fun action(request:
      ExternalRequest) {
    val receiver = request.argument(0)
    val ilf = IllocutionaryForce.parse
        (request.argument(1))
    val content = request.argument(2)
    val sender = request.sender
    sendMessage(receiver, Message(
        sender, ilf, content))
  }
}
```

External actions defined in this way can be imported in environments' `actions` blocks, similarly to actions defined on the fly, and invoked as any other action by agents:

```
agent("sender"){
  actions { action(Send) }
  plans {
    +achieve("greet_twice") then {
      execute("send"("receiver", "tell
          ", "hello!"))
      Send("receiver", "tell", "hello!
          ")
    }
  }
}
```

Message-passing

In JaKtA, messages are exchanged through the environment, message passing capabilities are modelled as external actions. Thus, JaKtA requires all environment implementations to handle agents' inboxes and dispatch messages, which enables customisation of message-passing protocols (including distributed ones) through the construction of personalised environments.

Technically, all environment implementations must support the following functionalities:

1. `sendMessage`, which delivers a message to a specific recipient agent, and
2. `broadcastMessage`, which delivers the message to all agents in the MAS.

In both cases, the message consists of triplet containing:

1. the name of the sender agent,
2. an illocutionary force (ILF) tag, which specifies how the message should be interpreted by the recipient, and
3. the payload, i.e., the actual content the sender wants to communicate—in the general case, a logic term.

The syntax is exemplified in the following example:

```
environment {
    actions {
        action("talkWith", ⟨Arity⟩) {
            sendMessage(⟨Recipient⟩,
            ⟨Message⟩) }
        action("helloEveryone", ⟨Arity⟩)
            { broadcastMessage(⟨Message⟩)
            }
    }
}
```

Once delivered to the recipients' message box, the message will be processed by the agent's reasoning cycle in compliance with the value of the ILF tag. JaKtA currently supports the following ILF values: `tell` and `achieve`, which – according to the KQML [22] specification – aim to share beliefs and delegating goals among agents, respectively. Summarising, the ILF tag impacts the way the recipient agent will process the message. The recipient agent reacts to the reception by means of a plan whose triggering event is the addition of a belief (`tell`) or a goal (`achieve`).

For instance, the following snippet exemplifies a MAS where a couple of agents enact one round of the ping-pong protocol. When the sender tags a message with the `tell` ILF, the intended effect is to add a belief to the recipient's belief base. The recipient then reacts similarly to how it would have reacted to the addition of a belief. The only difference lays in the fact that the belief, on the recipient side, is tagged with a logic structure denoting the source of the message—i.e., the sender agent's name.

```
mas {
    environment { actions { action(
        Send) } }
    agent("pinger") {
        goals { achieve("send_ping") }
        plans {
```

```
            +achieve("send_ping") then
                { Send("ponger", "
                tell", "ping") }
            +"pong".fromAnyone then {
                Stop }
        }
    }
    agent("ponger") {
        plans {
            +"ping".source(X) then {
                Send(X, "tell", "pong"
                ); Stop }
        }
    }
}
```

Notably, beliefs' source tags can be specified through the `.source()`, `.fromSelf`, or `.fromAnyone` methods.

AOP implementation of OOP/FP specifications

In the following example, we showcase how blended paradigms can be leveraged to implement a portion of a software system using the most appropriate tool for the job. In particular, we implement a Kotlin function (collatz) which verifies the Collatz conjecture [35] for the provided number. Without using any FFI, we implement the function by means of a MAS specification, in a way completely transparent to the caller.

```
fun collatz(number: Int) = mas {
  agent("collatz-agent") {
    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!");
            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)) }
    }
  }
}
```

This sample serves as an example of how JaKtA can be used to implement even small portions of large systems using AOP, if convenient.

Under the Hood of JaKtA

JaKtA is a full fledged, AgentSpeak(L)-compliant BDI technology. As such, it comes with its own BDI execution engine, giving semantics to the DSL. The choice of realising a fresh implementation of a BDI execution engine instead of reusing an existing one was driven by two major design goals:

1. to explore paradigm blending of AOP—and in particular BDI—with mainstream programming languages, and
2. to support modularity and pluggability of any aspect involving the execution of BDI systems—there including reasoning capabilities, message passing mechanisms, concurrency models, and the like.

Accordingly, the execution engine of JaKtA was designed and implemented from scratch to decouple agent specifications and their execution.

Architecturally, the JaKtA framework is composed by three main modules, namely:

1. the DSL module, which defines the syntax of the language;
2. the BDI interpreter, which governs the execution of agents and environments, regardless of the particular syntax used to define them; and
3. the concurrency management module, which regulates runtime, concurrency, and scheduling aspects for any system run by the BDI interpreter.

As depicted in Fig. 1, the three modules are inter-dependent in a layered way: the DSL module is built on top of the BDI interpreter, which in turn is built on top of the concurrency management module.

The DSL module is separate from the BDI interpreter module as it implements one possible syntax of many for BDI MAS specification. Other languages could be plugged in the same BDI interpreter, for instance, the Jason's parser can be, in principle, plugged on top of JaKtA's BDI interpreter, using the latter as engine. Similarly, Scala developers may design a different internal DSL and plug it on top of the existing BDI interpreter, realising in shorter time a way to write BDI agents in Scala. Summarising, the neat modularisation of DSL and BDI interpreter opens to many possible BDI agents implementations, of which the current JaKtA DSL is a possible instance.

The DSL Module

The DSL module contains the syntactic constructs described in "Overview of JaKtA's syntax". Inside this module, BDI abstractions from the interpreter module are mapped into the

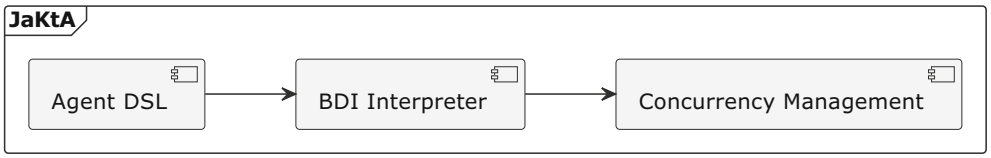


Fig. 1 JaKtA modules and their relations. Arrowheads indicate dependencies (depends-on relationship)

set of Kotlin classes and functions enabling users to write MAS specifications in Kotlin.

The BDI Interpreter Module

The BDI interpreter is the component where the many abstractions of the BDI model are reified into actual code, giving semantics to the DSL defined in "Overview of JaKtA's syntax". Put it simply, the BDI interpreter module provides interfaces and implementations for the many abstractions involved in the execution of a MAS, there including the ones which are not visible in the DSL syntax. It provides ways to extend and inject abstractions which are meant to be personalised by MAS developers (such as actions, or environments), and it encapsulates the abstractions which are not meant to be customised (e.g., the agents' control loop), as per the best practices in the field [36].

Main abstractions In particular, the BDI interpreter module provides interfaces and default implementations for the following abstractions:

- beliefs and belief bases,
- goals,
- events and event pools,
- plans and plan libraries,
- intentions and intention pools, and
- side-effects (i.e., the admissible consequences of actions),
- (internal and external) actions,
- environments,
- agents, and
- message passing mechanisms.

In this case, the module prescribes what each abstraction may (or may not) do, hence simultaneously enabling and constraining MAS developers' customisations. Most notably, all these abstractions are designed with immutability as a core design principle, including those whose state must vary over time through the *copy-on-write* strategy. This is key to support inspectability and reproducibility of the MAS execution, as well as to ease the introduction of parallelism in the execution of the MAS.

Agent lifecycle and MAS

Among the many abstractions provided by the BDI interpreter module, the most important ones are the *agent lifecycle* and the *MAS*. These are where all other abstractions are combined together to support the execution of BDI systems. In particular, the agent lifecycle is the component that dictates the execution of each single agent, whereas the MAS is the component which orchestrates the execution of all agents in the system, as well as their interaction with (and through) the environment.

As far as the agent lifecycle is concerned, this is the component that defines the control loop of each agent. Assuming the internals of each agent include:

1. a belief base,
2. a plan library,
3. a set of internal actions,
4. an event pool, and
5. an intention pool

the default implementation of the agent lifecycle prescribes that each iteration of each agent's control loop should encompass the following steps:

1. percepts and messages for the current agent are collected from the environment;
2. the belief base is revised according to the new percepts and messages;
3. events corresponding to added/removed beliefs or goals are generated and added to the current agent's event pool;
4. one event is extracted (and removed) from the event pool;
5. relevant plans for the event are selected from the current agent's plan library;
6. the guard of each selected plan is evaluated against the current agent's belief base;
7. one plan is selected for execution among the ones whose guard is satisfied;
8. the plan is assigned to either a new or pre-existing intention in the current agent's intention pool;
9. one intention is selected for execution in the current agent's intention pool;
10. the next operation in the top-most plan of the selected intention is executed, and then discarded;
11. any side-effect produced by the execution of the operation is collected and returned as output of the single iteration of the control loop.

Most notably, steps 2, 4, 5, 7, and 9 are designed to be customisable by users. It is also with mentioning that step 10

is where the execution of internal or external actions occurs. So, this is where the execution of custom OOP/FP code is performed. As a byproduct, actions may provoke side-effects, whose management is described in the following.

The MAS component is where the lifecycles of all agents in a system are tied together and coordinated, along with any computational activity involving the environment (e.g., message dispatching). Put it simply, this component is in charge of keeping track of which and how many agents compose the system. For each of these, the MAS shall govern the unlimited repetition of their control loop, following the aforementioned steps. In which order the many control loops of the many agents are interleaved is a matter of implementation—and several implementations are available in JaKtA, as enabled by the concurrency management module (cf. "The Concurrency Management Module").

Regardless of the particular order by which agents' control loops interleave, the MAS is responsibile to ensure the reification in the environment of any side-effect produced at each iteration of any agent's control loop. There, side-effects essentially consist of message exchanges among agents, or changes in the environment's data. Of course, the actual semantics of side-effects reification depends on the particular implementation of the environment.

Environment and message-passing mechanism

The environment is a first-class abstraction in JaKtA. However, the notion of environment is highly dependant on the particular application domain at hand. Hence, we designed a minimal environment which developers may customise depending on their need with the following responsibilities:

- keeping track the agents in the system;
- governing the perception of each agent by deciding which percepts to deliver to each agent;
- governing the actuation of each agent by making external actions available to each agent;
- governing the communication among agents by deciding how messages are delivered to each agent;
- enabling the stigmergic interaction among agents by making the environment's data available to each agent.

Of course, JaKtA also provides a default implementation of the environment, very simple and general purpose, where agents can perceive knowledge, act on its state, or send messages to other agents—assuming that other agents are running inside the same OS process.

To support inter-process or inter-machine communication, users must provide their custom environment implementation. This design choice was mainly driven by flexibility, especially related to network concerns; in fact, different users may be willing to leverage different communication protocols.

Actions and side-effects

Just as for other notable BDI languages, agents' specification in JaKtA are *declarative*, except in the case of (internal and external) actions.

To agents, actions are—conceptually—the way to make things happen (either in the environment or on themselves). However, "making things happen" may imply expressing computations which are not convenienty represented via declarative constructs. There, actions aim to bridge the gap between the declarative world of BDI agents and mainstream programming constructs. Thus, actions are the means by which agents can declaratively invoke functional, object-oriented, or imperative code.

Technically speaking, actions are arbitrary functionalities, written in Kotlin, which may provoke side-effects—either on the agent itself, or on the environment. To support this feature, the BDI interpreter module provides ad-hoc type hierarchy, represented in Fig. 2. Actions are functional interfaces triggered by a request (coming from an agent) and producing a response in return. Each action has a signature, which denotes the name and arity by which it is known to agents. Each request contains a reference to the triggering agent, and a list of actual arguments. Each response contains a list of side-effects, each one describing an edit to be performed on the agent's state, or on the environment. Admissible changes to the agent's state include: addition or removal of beliefs, goals, plans, or intentions, as well as the stopping or pausing of the agent itself. Admissible changes to the environment include: addition or removal of agents, update of data in the environment, or sending of messages to other agents. The way by which the action produces the response is up to the action implementors, which may exploit Kotlin to do whatever they want.

The Concurrency Management Module

The concurrency management module aims at letting users customise the concurrency model of their MAS. To put it simply, the module lets users choose *how* agents are scheduled and executed—i.e., how they can be allocated on top of threads. One may for instance choose to have a single thread per agent, or to have a thread pool shared by all agents, depending on the application at hand. In the second case, implementers may also choose the atomicity of agents' control loops: they may for instance make the single iteration of the loop atomic, hence allowing multiple agents interleave their execution in the most fine-grained way possible. The module also takes care of virtualising relevant aspects related to agents execution, such as the flow of time. For the sake of conciseness, we do not details in this work all the concurrency models supported by JaKtA's BDI interpreter; the interested reader may refer to the documentation of the framework for further details [37].

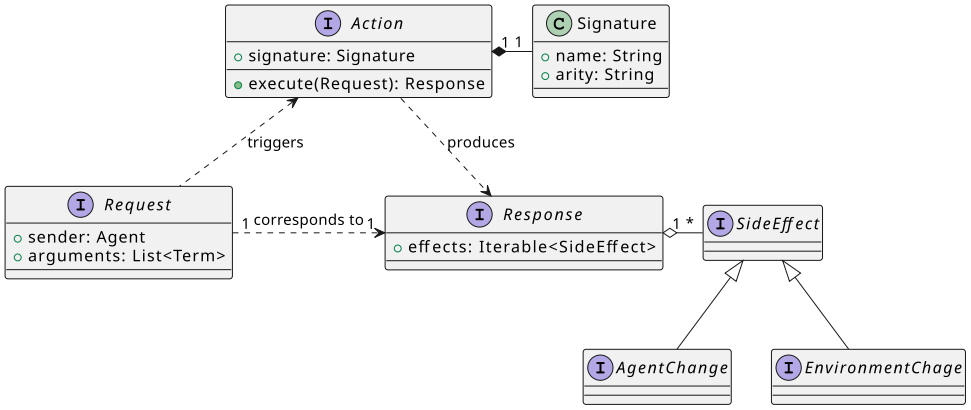


Fig. 2 JaKtA Action interface hierarchy

One key take-away here is that the BDI interpreter is built on top the concurrency module. Decoupling the BDI interpreter and the concurrency modules is based on the idea that the former decides *what* to execute, leaving to the latter the decision on *how* to execute it.

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.

Learning JaKtA

For the sake of conciseness, we keep the example deliberately minimal. The full code of the example is available on a public repository.[12] In the same repository, the reader can find several running examples of JaKtA applications, which can be used as a starting point to learn the framework. Since JaKtA is Kotlin-internal DSL, knowledge of the host language is needed to be proficient with the framework. For those who do not know Kotlin, we recommend to first learn Kotlin through the official documentation,[13] exercising with the Kotlin Koans Online,[14] then move to our example repository to learn JaKtA.

Reference Scenario

We selected a scenario to highlight the benefits of paradigm blending: we want to write a multi-agent modelling a Tic-TacToe match played on a $N \times N$ board, where N is only known at runtime. 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 N^{th} cell in the same direction is empty, write your mark in that cell; (iii) put a cross in random empty cell.

There are four alignment 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 ⟨*alignment*⟩ can be: `vertical`, `horizontal`, `diagonal`, and `antidiagonal`, while ⟨*dx*⟩, ⟨*dy*⟩ 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:

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

[13] https://kotlinlang.org/docs/home.html.

[14] https://play.kotlinlang.org/koans/overview.

```
+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:

```
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`. The `Put` action is defined as a Kotlin object, in another file, and extends the `AbstractExternalAction` provided by the framework:

```
object Put : AbstractExternalAction(
    Signature("put", 3)) {
  override fun action(request:
      ExternalRequest) {
    val x = request.argument(0)
    val y = request.argument(1)
    val mark = request.argument(2)
    updateData(mapOf("cell" to
        Triple(x, y, mark)))
  }
}
```

The action takes three arguments, which are the position where to insert the mark and the mark itself. Finally, it update the environment's data, by the means of the `updateData` environment's side effect.

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)).fromSelf
      impliedBy second }
  rule { name(listFrom(first, second,
      last = W)).fromSelf.impliedBy(
      first,
      second,
      (X - A) arithEq dx,
```

```
        (Y - B) arithEq dy,
        name(listFrom(second, last = W))
            .fromSelf,
    )
  }
}
```

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) =
  allPossibleCombinationsOf(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.

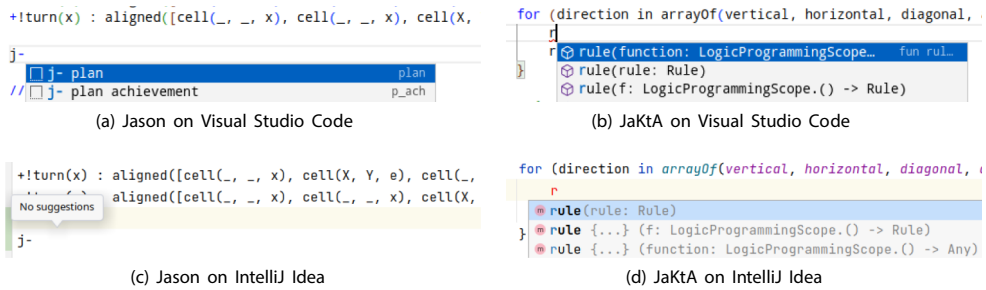
Tooling and Ecosystem

An indirect benefit of internal DSLs is the availability of inheriting the rich ecosystem of tools of the host language. We quickly exemplify in Fig. 3 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. Oppositely, JaKtA is fully supported in any IDE featuring Kotlin support (at the time of writing, this includes VSCode, Idea, Android Studio, Eclipse, and Atom[15]).

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,[16] as demonstrated by Fig. 4. JaKtA is available on Maven Central, and can thus be imported as an

[15] https://kotlinlang.org/docs/kotlin-ide.html .

[16] Code available at: https://github.com/jakta-bdi/jakta-android-example.



(a) Jason on Visual Studio Code (b) JaKtA on Visual Studio Code

(c) Jason on IntelliJ Idea (d) JaKtA on IntelliJ Idea

Fig. 3 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

Fig. 4 The TicTacToe MAS running on Android

ordinary dependency in any Android project, at the cost of a single line in the projects' Gradle build file.

Conclusion, Limitations, and Future Work

In this paper, we introduced JaKtA: an internal Kotlin DSL for BDI agent programming that strives to achieve true paradigm blending of AOP, OOP, and FP within a mainstream general-purpose programming language. We show how JaKtA can be used to implement a simple BDI agent and how paradigm blending can be used to achieve improved modularity and reusability, including 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 syntax or adopt new tools.

JaKtA is thus a first step towards the full integration of AOP in a modern programming language, achieving paradigm blending. This research direction opens up many interesting problems that can be explored going forward: in this section, we will thus start from the current limitations of JaKtA to discuss future work.

DSL Syntax

Since the target users of JaKtA are both experienced agent developers familiar with the BDI style of programming and novice programmers coming from traditional OOP languages, when designing the DSL syntax for JaKtA we tried to balance between familiarity with existing agent technologies and expressiveness.

Given its popularity in the community, we draw inspiration from Jason, producing a syntax with several similarities. The differences are partly deliberate and partly consequential to the syntactic boundaries imposed by the host language. For instance, only a small subset of symbols representing operators in Kotlin can be customized: as an example, while the symbol `!` is an overridable operator, the symbol `?` cannot be used nor defined.

At the same time, we believe these limitations allowed us to use explicit keywords such as `achieve` and `test` over dedicated symbols that could be considered cryptic for novice users. More generally, in many cases, we had to pick syntactical design choices harmonizing the Jason's syntax with the Kotlin one. As a result, the JaKtA syntax is more verbose compared to Jason's, but we believe that this is a reasonable trade-off to get the benefits of using Kotlin as hosting language.

As the DSL module is independent of the interpreter, we plan to experiment with the syntax in order to refine the current version and make it more pleasant to use. We will thus be open to collecting feedback from other research groups experimenting with JaKtA as well as from students approaching AOP for the very first time.

We also believe that the freedom given by the JaKtA DSL implementation could allow for different *dialects* to co-exist, building on the same interpreter. This is especially interesting when considering porting JaKtA to different platforms other than the JVM, a matter deserving its own paragraph.

Kotlin multiplatform support

Although the multiplatform targeting supported by Kotlin is among the reasons why the technology was chosen, the current implementation of JaKtA is available only for the JVM.

In the future, we plan to fully leverage this capability, thus enabling the exploitation of a single language and interpreter for running BDI systems inside web browsers (Kotlin/JavaScript), mobile (Kotlin/Android, Kotlin/Native for iOS), wearable and low-power (Kotlin/Native), and general-purpose (Kotlin/JVM) applications, retaining cross-platform interoperability.

This would make JaKtA a desirable solution for emerging scenarios with heterogeneous devices, especially when this feature is combined with the freedom of customisation given by the fact that most architectural components can be swapped with custom implementations.

Paradigm blending

Approaching the interesting problem of paradigm *blending*, we mainly considered the benefits that mixing OOP and FP constructs could bring to the development of multi-agent systems.

There are, however, some limitations in the way JaKtA handles blending that might be interesting to address with further iterations on the language. For instance, some BDI languages such as Jason and Astra introduce syntactic constructs akin to those of imperative languages (e.g., `for`, `foreach`, and `if` statements) to ease the definition of procedural plans, as an attempt to become more developer-friendly. These constructs are not part of the common formalization of BDI languages. They are *syntactic sugar* aimed to reduce the verbosity of plans, or the need to define auxiliary actions.

As the JaKtA DSL is embedded in the Kotlin language, agents specifications – and, in particular, plans definitions – may contain Kotlin's imperative constructs too—such as `for` loops, and `if` or `when` statements (cf. "Multi-paradigm integration and meta-programmability" for an example on *meta-programmability*). However, this may produce surprising behaviours in the eyes of inexperienced developers. In fact, in our DSL, any piece of Kotlin code just aims at *building* the declarative specification of a MAS—there including agents and plans. Yet, such specification may only contain AgentSpeak(L)-compliant, declarative constructs, as those are the only ones the JaKtA interpreter supports. The code used within a MAS definition is executed only after the MAS structure has been assembled, not when the interpreter runs the MAS itself. Although this is arguably a desirable effect, we reckon that it may be astonishing for novice developers,

Along this line, it would be interesting to explore whether JaKtA's BDI interpreter can be extended to natively support as one further step towards full feature equivalence with regard to Jason or Astra.

At the same time, in this work, we do not explore what benefits an AOP portion of code could bring to the development of an OOP/FP project. We believe that, to achieve true blending, it might be interesting to explore this option more in depth; as it could bring to even more intertwined connections among paradigms. For instance, a bunch of agents may be employed to cooperatively execute a parallelisable task, whose result may be returned asynchronously to the OOP/FP world (e.g., via promises). Despite being technically straightforward, such a functionality would come with many challenging theoretical implications—e.g., when should each agent terminate? when should the MAS terminate? what is the best way to propagate results back to the OOP/FP realm? At the same time, we believe features of this kind might strengthen the link between paradigms, and truly support *blending*, giving even more choices to developers in picking the paradigm, and the abstraction, they find themselves more comfortable with for each specific task.

Debugging

Among the main benefits of choosing an internal DSL, we mentioned the possibility of reusing the tools available for the host language without the need to implement new ones from scratch. This includes debugging tools, which are essential for robust software engineering.

While JaKtA programs can already be debugged through the existing the Kotlin debuggers, available for many mainstream IDE, the abstractions observed inside the inspector are those internal to Kotlin. This means, for instance, that when stopped on a breakpoint the tool will show Kotlin objects and references instead of JaKtA's higher-level AOP constructs.

Moreover, using breakpoints is trickier than in ordinary Kotlin, a breakpoint positioned without care might be triggered at a surprising moment (typically, before the MAS is even started). This is a consequence of the strategy used to construct the DSL: we heavily rely on lambda expressions whose code is evaluated at MAS *construction* to produce an executable system whose actual execution is performed later by the interpreter, so breakpoints set on those lines will be triggered at construction time—indeed, it is not a coincidence that internal DSLs in Kotlin are called "type-safe builders".

A third issue that makes the native Kotlin debugger suboptimal concerns the stack inspection. As the abstractions visualised in the tool are those of the host language, so the stack frames will be those native of Kotlin, producing stacktraces harder to inspect than those mapping AOP abstractions directly.

Debugging of BDI agents is indeed an open issue in the community, different platforms adopt different approaches; as an example, Jason famously offers a graphical interface for a *mind inspector* showing the current beliefs and intentions that are present in the agent's "mind" during execution.

We believe that, although initial debugging support in the form of the native Kotlin debugger is useful, it might be interesting to investigate how to support the developers producing more meaningful error messages and dedicated debugging tools that preserve the abstractions of the agent program.

Concurrency management

Concerning runtime behavior, JaKtA's architecture has been designed to separate the concurrency model from the agent specification.

Currently, we only support a handful of alternatives, including the widely used 1A1T execution strategy that runs agents concurrently on separate threads (one per agent), as well as AA1T, where the whole MAS runs on a single thread. As the latter may be useful for debugging and simulation scenarios, JaKtA also supports a virtualised-time mode, where the flow of time is detached from the real time.

Accordingly, the concurrency module of JaKtA enables exploring and comparing the impact of concurrency abstractions on the design of a MAS. Along this line, we plan to (i) widen the set of execution strategies supported by JaKtA, possibly including more advanced ones such as AA1E, and to (ii) investigate how JaKtA can be integrated with mainstream simulation and concurrency frameworks to provide better support to the development of simulated or parallel MAS.

Distributed Multi-Agent Systems and Messaging

Many applications in the field of MAS are distributed systems, where agents move or communicate across networked computers. At the time of writing, though, JaKtA does not support agents communicating or moving across the network, as the only message-passing mechanism currently implemented is in-process.

Although we acknowledge this limitation, JaKtA already supports full customisation of the message-passing mechanism—a feature that we plan to exploit soon by including a *distributed* message-passing mechanism in the default distribution of our framework.

The same goes for the limited set of performatives that are currently supported: a richer set is fundamental to support the adoption of JaKtA by MAS developers; our goal mid-term is to support the same set of performatives supported by Jason.

In the long run, we may also modularise the ACL supported by JaKtA, paving the way towards the possibility of supporting multiple ACL.

Action libraries

For any new programming framework, the surrounding ecosystem of libraries is essential to thrive. JaKtA benefits from the possibility of integrating easily with the Kotlin ecosystem and libraries.

At the same time, we understand that MASs developers may benefit from the existence of a rich standard library of internal actions, as well as from sets of easily importable and reusable external actions for specific problems (e.g., integrating with Web services or databases).

Creating and distributing such libraries of actions is trivial in JaKtA, as developers can directly reuse the build tools and software repositories commonly used in the development of vanilla Kotlin programs, such as Gradle and Maven Central.

Currently, JaKtA is shipped with a very minimal core set of features that we plan to expand over time. Moreover, JaKtA is an open-source project: we warmly welcome contributions, including those in the form of libraries, and we plan to collect and promote projects affiliated with and extending JaKtA, hence encouraging the creation of an active community.

Performance evaluation and optimization In this paper, our focus is on a software engineering perspective, discussing the application of internal DSLs to BDI agent programming, demonstrating feasibility, design challenges, and benefits of the approach. However, for the tool to be practically useful, future work need to evaluate its performance in detail. We internally conducted some initial profiling, discovering that (expectedly) most of the performance burden is due to JaKtA's immutable-by-default design, which purposedly trades performance for thread-safety by leveraging on copy-on-write data structures—in order guarantee the agents' interpreter works correctly regardless of the concurrency model of choice. We believe that performance can still be heavily improved, thus, we are delaying performance measurement until a reasonably optimized version of the tool is ready. In future work, we will first analyze in details and tackle the performance bottlenecks of JaKtA, then run benchmarks to compare the performance of JaKtA with other BDI tools, primarily Jason.

Author Contributions The contribution in this paper can be summarized as follows: • *Martina Baiardi*: Conceptualization, Data curation, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, and Writing—review & editing. • *Giovanni Ciatto*: Conceptualization, Investigation, Methodology, Software, Supervision, Validation, Writing—original draft, and Writing—review & editing. • *Samuele Burattini*: Conceptualization, Validation, Writing—original draft, and Writing—review & editing. • *Danilo Pianini*: Funding acquisition, Resources, Supervision, Validation, Writing – original draft, and Writing—review & editing.

Funding Open access funding provided by Alma Mater Studiorum - Università di Bologna within the CRUI-CARE Agreement. This work has been partially supported by: • "WOOD4.0 - Woodworking Machines for Industry 4.0", Emilia-Romagna regional project, call 2022, art. 6 L.R. N. 14/2014;
• "FAIR-Future Artificial Intelligence Research", Spoke 8 "Pervasive AI" (PNRR, M4C2, Investimento 1.3, Partenariato Esteso PE00000013), funded by the European Commission under the NextGenerationEU programme; and
• "Digital Twins Ecosystems for the clinical, strategic and process governance in Healthcare" (PNRR, M4C2 Investment 3.3 DM 352/2022 - CUP J33C22001400009) funded by the European Union - NextGenerationEU and Azienda Unità Sanitaria Locale (AUSL) della Romagna.

Data Availability Not Applicable.

Declarations

Conflict of interest The authors declare they have no financial or personal relationship with any third party whose interests could be positively or negatively influenced by the article's content.

Research Involving Human and/or Animals Not applicable.

Informed Consent Not applicable.

References

1. Leroy X, Doligez D, Frisch A, Garrigue J, Rémy D, Vouillon J. The ocaml system: documentation and user's manual. INRIA. 2021;3:42.
2. Mazinanian D, Ketkar A, Tsantalis N, Dig D. Understanding the use of lambda expressions in java. Proc ACM Program Lang. 2017;1(OOPSLA):85–18531. https://doi.org/10.1145/3133909.
3. Riti P. Practical scala DSLs: real-world applications using domain specific languages. Berkeley: Apress; 2018. https://doi.org/10.1007/978-1-4842-3036-7.
4. Odersky M, Rompf T. Unifying functional and object-oriented programming with scala. Commun ACM. 2014;57(4):76–86. https://doi.org/10.1145/2591013.
5. Rao AS, Georgeff MP. Modeling rational agents within a bdi-architecture. In: Allen JF, Fikes R, Sandewall E. editors. Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91), 1991; pp. 473–484. Morgan Kaufmann, Cambridge, MA, USA.
6. Bordini RH, Hübner JF, Wooldridge MJ. Programming Multi-Agent Systems in AgentSpeak Using Jason, p. 292. John Wiley & Sons, Ltd, USA 2007; http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470029005.html. Accessed 27 Sept 2024.
7. Collier RW, Russell SE, Lillis D. Reflecting on agent programming with AgentSpeak(L). In: PRIMA 2015: Principles and Practice of Multi-Agent Systems. Lecture Notes in Computer Science, 2015; pp. 351–366. Springer, Bertinoro, Italy. https://doi.org/10.1007/978-3-319-25524-8_22.
8. Hindriks KV. Programming rational agents in GOAL. In: El Fallah Seghrouchni A, Dix J, Dastani M, Bordini RH, editors. Multi-agent programming: languages, tools and applications. Boston: Springer; 2009. p. 119–57. https://doi.org/10.1007/978-0-387-89299-3_4.
9. Baiardi M, Burattini S, Ciatto G, Pianini D. Jakta: BDI agent-oriented programming in pure kotlin. In: Malvone V, Murano A. editors. Multi-agent systems—20th European Conference (EUMAS 2023). Lecture Notes in Computer Science, 2023; vol. 14282, pp. 49–65. Springer, Naples, Italy. https://doi.org/10.1007/978-3-031-43264-4_4.
10. Stroustrup B. The C++ programming language. 3rd ed. Addison-Wesley; 1997.
11. 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. editors. 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 2020; 21st Workshop "From Objects to Agents" (WOA 2020), Bologna, Italy, 14–16 September 2020. Proceedings. http://ceur-ws.org/Vol-2706/paper14.pdf
12. Bratman M, et al. Intention, plans, and practical reason, vol. 10. Cambridge: Harvard University Press; 1987.
13. Cohen PR, Levesque HJ. Intention is choice with commitment. Artif Intell. 1990;42(2–3):213–61. https://doi.org/10.1016/0004-3702(90)90055-5.
14. Omicini A, Ricci A, Viroli M. Artifacts in the a&a meta-model for multi-agent systems. Auton Agents Multi Agent Syst. 2008;17(3):432–56. https://doi.org/10.1007/s10458-008-9053-x.
15. Ricci A, Piunti M, Viroli M. Environment programming in multi-agent systems—an artifact-based perspective. Auton Agent Multi-Agent Syst. 2011;23(2):158–92. https://doi.org/10.1007/s10458-010-9140-7.
16. Georgeff MP, Lansky AL. Reactive reasoning and planning In: AAAI. 1987;87:677–82.
17. Omicini A. Agents writing on walls: cognitive stigmergy and beyond. In: Paglieri F, Tummolini L, Falcone R, Miceli M, editors. The goals of cognition. Essays in Honor of Cristiano Castelfranchi, vol. 20. Tributes. London: College Publications; 2012. p. 565–78 (**Chap. 30**).
18. Hadeli K, Valckenaers P, Kollingbaum MJ, Brussel HV. Multi-agent coordination and control using stigmergy. Comput Ind. 2004;53(1):75–96. https://doi.org/10.1016/S0166-3615(03)00123-4.
19. Weyns D, Omicini A, Odell J. Environment as a first class abstraction in multiagent systems. Auton Agents Multi Agent Syst. 2007;14(1):5–30. https://doi.org/10.1007/S10458-006-0012-0.
20. Omicini A, Ricci A, Viroli M, Castelfranchi C, Tummolini L. Coordination artifacts: Environment-based coordination for intelligent agents. In: 3rd International Joint Conference on autonomous agents and multiagent systems (AAMAS 2004), 2004; pp. 286–293. IEEE Computer Society, New York, NY, USA. https://doi.org/10.1109/AAMAS.2004.10070.
21. Chaib-draa B, Dignum F. Trends in agent communication language. Comput Intell. 2002;18(2):89–101. https://doi.org/10.1111/1467-8640.00184.
22. 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. CIKM '94, 1994; pp. 456–463. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/191246.191322.
23. Traum DR. Speech acts for dialogue agents. In: Wooldridge, M., Rao, A. (eds) Foundations of rational agency. Springer, Dordrecht, 1999. vol. 14. https://doi.org/10.1007/978-94-015-9204-8_1
24. Calegari R, Ciatto G, Mascardi V, Omicini A. Logic-based technologies for multi-agent systems: a systematic literature review. Auton Agent Multi-Agent Syst. 2021;35(1):1–1167. https://doi.org/10.1007/s10458-020-09478-3.
25. Palanca J, Rincon JA, Carrascosa C, Julián V, Terrasa A. A flexible agent architecture in SPADE. In: Dignum F, Mathieu P, Corchado JM, Prieta F. editors. Advances in practical applications of agents, multi-agent systems, and complex systems simulation. The PAAMS Collection - 20th International Conference, PAAMS. Lecture Notes in Computer Science, vol. 13616, pp. 320–331. Springer, L'Aquila, Italy 2022; https://doi.org/10.1007/978-3-031-18192-4_26.
26. D'Urso F, Longo CF, Santoro C. Programming intelligent iot systems with a python-based declarative tool. In: Savaglio C, Fortino G, Ciatto G, Omicini A. editors. Proceedings of the 1st Workshop on Artificial Intelligence and Internet of Things; Co-located with the 18th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2019). CEUR Workshop Proceed-

ings, 2019; vol. 2502, pp. 68–81. CEUR-WS.org, Rende (CS), Italy. https://ceur-ws.org/Vol-2502/paper5.pdf. Accessed 27 Sept 2024.
27. Winikoff M. JACKTM intelligent agents: an industrial strength platform. vol. 15, pp. 175–193. https://doi.org/10.1007/0-387-26350-0_7.
28. Pokahr A, Braubach L, Lamersdorf W. Jadex: a BDI reasoning engine. In: Håkansson A, Nguyen NT, Hartung RL, Howlett RJ, Jain LC, editors. Agent and multi-agent systems: technologies and applications. Lecture Notes in Computer Science, vol. 5559. Berlin: Springer; 2005. p. 149–74. https://doi.org/10.1007/0-387-26350-0_6.
29. Körner P, Leuschel M, Barbosa J, Costa VS, Dahl V, Hermenegildo MV, Morales JF, Wielemaker J, Diaz D, Abreu S. Fifty years of prolog and beyond. Theory Pract Log Program. 2022;22(6):776–858. https://doi.org/10.1017/S1471068422000102.
30. Bergenti F, Monica S, Petrosino G. A comprehensive presentation of the jadescript agent-oriented programming language. In: Malvone V, Murano, A. editors. Multi-agent systems- 20th European Conference, (EUMAS 2023). Lecture Notes in Computer Science, vol. 14282, pp. 100–115. Springer, Naples, Italy 2023; https://doi.org/10.1007/978-3-031-43264-4_7.
31. Busetta P, Howden N, Rönnquist R, Hodgson A. Structuring bdi agents in functional clusters. In: International Workshop on Agent Theories, Architectures, and Languages, 1999; pp. 277–289. Springer.
32. Johann P, Polonsky A. Higher-kinded data types: Syntax and semantics. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, pp. 1–13. IEEE, Vancouver, BC, Canada 2019. https://doi.org/10.1109/LICS.2019.8785657.
33. Baiardi Martina, Ciatto G, Pianini D. Semantic Release Bot: jakta-bdi/jakta: v0.3.0. Zenodo 2023. https://doi.org/10.5281/zenodo.7900584.
34. Ciatto G, Calegari R, Omicini A. 2P- Kt: a logic-based ecosystem for symbolic AI. SoftwareX. 2021;16:100817–11008177. https://doi.org/10.1016/j.softx.2021.100817.
35. Andrei S, Masalagiu C. About the Collatz conjecture. Acta Inform. 1998;35(2):167–79. https://doi.org/10.1007/s002360050117.
36. Dennis LA, Fisher M, Webster MP, Bordini RH. Model checking agent programming languages. Autom Softw Eng. 2012;19(1):5–63. https://doi.org/10.1007/S10515-011-0088-X.
37. Baiardi M. JACOP: Programming BDI agents with pluggable concurrency model. Master's thesis, University of Bologna 2023.