Cross-Paradigm Interoperability Between Jadescript and Java

Giuseppe Petrosino[1], Stefania Monica[1] and Federico Bergenti[2]
[1]*Dipartimento di Scienze e Metodi dell'Ingegneria, Università degli Studi di Modena e Reggio Emilia, Reggio Emilia, Italy*
[2]*Dipartimento di Scienze Matematiche, Fisiche e Informatiche, Università degli Studi di Parma, Parma, Italy*

Keywords: Agent-Oriented Programming, Jadescript, JADE.

Abstract: Jadescript is a recent language for practical agent-oriented programming that aims at easing the development of multi-agent systems for real-world applications. Normally, these applications require extensive and structured reuse of existing software modules, which are commonly developed using object-oriented or legacy technologies. Jadescript has been originally designed to ease the translation to Java and, as such, it natively eases the interoperability with Java, and therefore, with mainstream and legacy technologies. This paper overviews the features that have been recently added to Jadescript to support effective two-way interoperability with Java. Moreover, this paper thoroughly discusses the main ideas behind such features by framing them in a comparison with related work, and by outlining possible directions for further developments.

1 INTRODUCTION

Programming language interoperability is a key aspect of modern software development because it allows software components written in various languages to interact and share data. It provides several advantages when designing a new system or extending the functionality of existing systems, like platform adaptation and code reuse. The rewriting of code in a new language to solve previously addressed subproblems is cumbersome, and it represents an obstacle to the adoption of new languages. Moreover, languages that operate at a higher level of abstraction often pay inherent costs, e.g., in terms of worse run-time performance. The interoperability with languages at a lower level of abstraction, which can better express the details of their low level of abstraction, can mitigate such issues. As a matter of fact, it is generally accepted that the most appropriate programming language is different for each situation. This is because each language can represent better, or worse, certain aspects of the addressed problems and of the programs that are proposed as their solutions.

Jadescript (Bergenti and Petrosino, 2018; Bergenti et al., 2018; Bergenti et al., 2020) is an *Agent-Oriented Programming* (*AOP*) language. Its main abstractions derive from agent technology, and therefore it can be used effectively to model software systems around software agents. Jadescript agents are based on JADE (Bellifemine et al., 2005), a Java framework with a long history (Bergenti et al., 2020) in the development and administration of *Multi-Agent Systems* (*MAS*). JADE is considered the de facto reference implementation of the *Foundation for Intelligent Physical Agents* (*FIPA*) specifications. Jadescript is designed to be translated to Java, it is supported by a static type system, and it adopts an event-driven programming style. Jadescript provides language facilities to define agents, agent behaviours, and communication ontologies with a syntax inspired from agent pseudocode. The development of Jadescript, and of its companion development tools, has recently reached maturity with several advanced features (Petrosino and Bergenti, 2019; Petrosino et al., 2021; Petrosino et al., 2022b) that make it a complete tool to program MASs. However, the development of agent-based software to address real-world problems, like the ones encountered in industrial applications of agents (Bergenti et al., 2015), requires Jadescript agents to interact with existing software. This is required to deal with legacy systems, for which agents-based approaches have been successfully applied (Genesereth and Ketchpel, 1994), to create graphical interfaces to improve the interactions with users, or to access external software or hardware (Iotti et al., 2020). One approach that is readily available to achieve Jadescript-Java interoperability uses the interaction capabilities of agents and the fact that both Jadescript and JADE agents are FIPA agents. Thanks to this, they can use standardized

Petrosino, G., Monica, S. and Bergenti, F.
Cross-Paradigm Interoperability Between Jadescript and Java.
DOI: 10.5220/0011619300003393
In *Proceedings of the 15th International Conference on Agents and Artificial Intelligence (ICAART 2023) - Volume 1*, pages 165-172
ISBN: 978-989-758-623-1; ISSN: 2184-433X

speech acts and shared ontologies to communicate and coordinate. Therefore, the interaction between Jadescript agents and Java objects can be achieved at the agent level by designing specific JADE agents to act as mediators or wrappers of Java objects. Such an approach remains valid, but some drawbacks in common situations are easily identified. One drawback is that the construction of mediator agents to let Jadescript agents execute simple procedures written in Java might constitute a development cost that does not scale well to large projects. Moreover, this approach adds considerable runtime overhead to the execution of the target procedures, making Jadescript agents unavoidably slow. On the other hand, the introduction of a dedicated interoperability support in Jadescript, together with a specific Java API, can significantly simplify the development process for the discussed scenarios. The main contribution of this paper is to discuss the recent changes introduced in Jadescript to let agents easily interact with existing code written in Java.

All the presented features introduced in Jadescript to support effective interoperability with Java were designed to satisfy a set of specific requirements. First, the features should not excessively increase the complexity of the language. For example, from a syntactical perspective, most of the features are limited to the introduction of the `native` modifier, which can be prepended to some of the constructs already available in the language. The use of this modifier changes the meaning of the modified constructs in ways that are discussed in Sect. 2. Second, the design of each feature should attempt to minimize the amount of additional Java and Jadescript code required to achieve interoperability. Unfortunately, this glue code cannot be null in many cases, mostly because of the relevant differences between the two languages.

This paper introduces all the presented features in Sect. 2, which is followed by a discussion and a comparison with similar features of other AOP languages in Sect. 3. After a brief discussion of the main possibilities to further improve the presented interoperabiltiy features, Sect. 4 concludes the paper.

2 JADESCRIPT-JAVA INTEROPERABILITY

This section describes the design of the proposed support for the interoperability between Jadescript and Java. As briefly discussed earlier, the possibility for Jadescript to effectively interface the Java virtual machine and its libraries is of paramount importance to promote reuse of existing software modules.

2.1 Jadescript and Java Data Types

A recent paper (Petrosino et al., 2022a) provides a detailed description of the Jadescript type system, and it highlights that the data types adopted for Jadescript are peculiar for several reasons. Actually, Jadescript is an AOP language, and its data types considers abstractions that are strongly related to agents and MASs. Jadescript does not adopt the *Object-Oriented Programming* (*OOP*) paradigm, and the language purposely lacks the abstractions needed to manipulate classes and objects. In addition, Jadescript is a very high-level language, and its data types are part of the means that it provides to deal with data at the adopted high level of abstraction. Finally, Jadescript is designed to target JADE, which makes the language–and its data types–designed to work with the abstractions standardized by FIPA and adopted by JADE.

One of the main consequences of these peculiarities of the Jadescript type system is that a one-to-one relationship between Jadescript data types and Java data types is not possible. Such a relationship is a characteristic that several mainstream programming languages, e.g, Kotlin and Groovy, provide to offer a direct support for interoperability with Java. Its absence constrains the way the two languages can share data when some code written in either one of them reuses the code written in the other, and it ultimately causes two major problems. First, note that Jadescript is designed to be translated to Java, and therefore each one of its data types has a corresponding representation in Java, either as a primitive data type, or as a class or an interface. However, this might not be the case in the future, which is the first of the two problems. Actually, the growing number of features provided by Jadescript for agent programming might require the introduction of new data types that might not be properly represented in Java, and that, therefore, would be erased during the compilation process. Second, Jadescript is not an OOP language, and therefore no objects are available to Jadescript programmers, which causes the second problem because not all Java data types can be properly mirrored by Jadescript counterparts. In order to target both these problems, Jadescript provides the features described below to create Jadescript values in Java and to convert values from a Java data type to a Jadescript data type in a simple and effective way.

The set of interoperability features that has been recently added to Jadescript are not limited to changes in the language. Three sets of new utilities designed to write Java code intended to interoperate with Jadescript code are provided in the Jadescript runtime support, as described below.

First, to assist the conversion of values from Java data types to Jadescript, the `Jadescript` class, included in the Jadescript runtime support, exposes a set of overloaded public static methods named `valueOf`. Such methods take one parameter each, and they return, after applying suitable conversions, values compatible with Jadescript.

Second, in addition to the `valueOf` methods, the `Jadescript` class provides a set of methods called `as`*X*, where *X* is the name of one of the built-in Jadescript data types, e.g., `asInteger`, `asDuration`, and `asPerformative`. Each one of these methods accepts a parameter of type `Object`, and it returns, in Java, the conversions performed by the `as` operator in Jadescript. For example, expression `"PT1S" as duration` in Jadescript converts the text literal `"PT1S"`, formatted according the ISO 8601 standard, to a duration representing a duration of one second. The same operation can be performed in Java by invoking `Jadescript.asDuration("PT1S")`. Note that not all conversions from Java values to Jadescript values are possible, and when an invalid conversion is attempted, an exception of type `ConversionException` is thrown.

Third, some Jadescript data types (`duration`, `timestamp`, and `performative`) have Java counterparts represented by classes provided by the Jadescript runtime support. These classes provide methods to create values and execute operations on the corresponding data types in Java. For example, the Java expression `n.plus(Jadescript.asDuration("PT1M"))` produces a Jadescript timestamp that represents the time instant that follow `n` after one minute.

Finally, note that, when using the Jadescript-Java interoperability features described in the remaining of this paper, the conversion of a value from a Jadescript data type to a Java data type is automatic and completely handled by the compiler. Instead, the conversion of a value from a Java data type to a Jadescript data type is not automatic, and it requires dedicated glue code. However, when using an interoperability feature from Java code, the use values of the correct Jadescript data type is always enforced at compile-time, with the notable exception of native expressions and statements described in Sect. 2.

2.2 Creating Jadescript Agents in Java

Jadescript is supported by a set of tools designed to facilitate the development of agents. One of these tools is a plug-in for the Eclipse IDE that provides a development environment with useful features, e.g., syntax highlighting. The plug-in also includes a wizard that assists programmers to configure and launch JADE platforms and containers (Bellifemine et al., 2007) populated with the agents written in Jadescript. Another way to launch Jadescript agents is via the command line (Caire et al., 2010) by specifying a Java class generated by the Jadescript compiler for each one of the desired agents together with respective start-up arguments. However, in several applications, it is necessary to start containers and agents in Java. A set of new facilities provided of the Jadescript runtime support simplifies the usage in Java of agents written in Jadescript.

```
1 module music
2
3 agent MusicDownloader uses ontology Music
4   on create with title as text,
5       artist as text do
6     activate BuyTrack(artist, title)
```

Listing 1: An example of a toy Jadescript agent.

Remember that a JADE platform is composed of several connected containers. Each platform has exactly one *main* container, and several *peripheral* containers. All peripheral containers register to the main container at start-up by specifying the network address and the port of the main container. A, main or peripheral, container can be created in Java using the `Jadescript` class by invoking the public static methods named `newMainContainer` or `newContainer`, respectively. These methods accept the network address and the port of the main container together with a platform identifier. Both methods return a `ContainerController` instance, which can be used, e.g., to create agents or to destroy the container.

For each agent defined in Jadescript, the compiler generates a Java class with the same name that provides a static method to create and initialize the agent. This method accepts as parameters: the `ContainerController` instance of the container in which the agent will be created, a string that is used as the local name of the agent, and one parameter for each one of the parameters required by the `on create` event handler of the Jadescript agent.

For example, consider the `MusicDownloader` agent in Listing 1, whose primary job is to update a music library with a track (if missing in the library) specified at start-up by stating the title and the name of the artist.

The code in Listing 1 is just an example to illustrate the creation of agents from Java code, so the `BuyTrack` behaviour is omitted. Starting from this agent definition, the Jadescript compiler generates the Java class `music.MusicDownloader`, which contains a static method named `create`. This method accepts

```
public class MusicDownloaderMain{
  public static void main(String[] args){
    ContainerController mainContainer=
      Jadescript.newMainContainer();
    JadescriptAgentController downloader=
      music.MusicDownloader.create(
        mainContainer,
        "downloader",
        "Johann Sebastian Bach",
        "Cello Suite No.1, Prelude");
  }
}
```

Listing 2: The creation of a main container and a Jadescript agent in Java.

a `ContainerController` instance, the agent local name, and two additional strings, one for the track title and one for the artist name. This method returns a `JadescriptAgentController` instance, which provides a thread-safe interface to the created agent that can be used to issue lifecycle commands (e.g., to shutdown and remove itself from the platform) or to notify native events to the agent. Listing 2 shows an example of the use of the generated class from Java.

2.3 Native Events

Jadescript adopts an event-driven programming style. Agents and behaviours can include several event handlers that define how to react to various external (i.e., messages from other agents) or internal (i.e., changes of the internal state, failures, and exceptions) events. *Native events* are an additional kind of event recently introduced in the language. A Jadescript agent can listen to native events through its active behaviours, when they contain an applicable *native event handler*. The details about received native events are reported at runtime by predicates or atomic propositions, and the programmer can use ontology declarations to define schemas for them. The syntax for a native event handler (here referred as *NEHandler*) is:

⟨*NEHandler*⟩ ::= 'on' 'native' ⟨*Pattern*⟩? ('when' ⟨*Expr*⟩)? 'do' ⟨*CodeBlock*⟩

where *Pattern* is an optional pattern (Petrosino and Bergenti, 2019) for the predicate or the proposition describing the event, *Expr* after the `when` keyword is a boolean expression expressing additional preconditions, and *CodeBlock* is a section of procedural code that defines what to do when a matching event is selected for handling.

In the current version of Jadescript, native events can be notified only from Java code by using the method named `emit` on a `JadescriptAgentController` instance. This method accepts one proposition, which can be defined in Jadescript using `proposition` and `predicate` declarations in ontologies. Note that the Java code written to create containers and agents is executed on a different thread than the ones on which JADE (and consequentially, Jadescript) agents execute. However, events notified through a `JadescriptAgentController` instance are pushed at the end of a queue internal to the agent, in a thread-safe way. Actually, the mechanism that is used to push an event to an agent is based on the *Object to Agent* (*O2A*) facility (Bellifemine et al., 2007) that JADE provides to safely interact with agents.

2.4 Use of Java Methods in Jadescript

This subsection describes a set of features that is provided to directly access methods written in Java from Jadescript. To achieve this, two new constructs are introduced in the language, namely the `do native` statement, and the `native` expression.

The syntax of a `do native` statement (here referred as *DNStatement*) is:

⟨*DNStatement*⟩ ::= 'do' 'native' ⟨*MetID*⟩ ('with' ⟨*ArgList*⟩)?

where *Expr* is an expression, *Expr:Text* is an expression evaluating to a `text` value, *Identifier* is a Jadescript identifier, and *ArgList* is a comma-separated list of expressions. This statement can be used to invoke a Java static method identified by *MetID*, with the arguments enumerated in the *ArgList*. The method is resolved at compile time if *MetID* is a Java fully qualified name expressed as a sequence of identifiers separated by dots. In this case, the compiler can check the existence of the Java method in the environment and the conformance of the types of the arguments. On the contrary, if *MetID* is an expression evaluating to a `text` value, the method is resolved at runtime, using the obtained text as the fully qualified name of the requested method. Java methods resolved at runtime are invoked in the generated code using the Java Reflection API. In this second case, the compiler does not check the conformance of the types of the arguments, and a Jadescript exception (Petrosino et al., 2022b) is thrown if the types of the arguments are not compatible with the parameters of any of the resolved methods. Note that, in both mentioned cases, as mentioned in Sect. 1 and Sect. 2, all data types used in the signature of the requested method must be compatible with Jadescript data types. Moreover, note that the `do native` construct is a statement, and therefore it cannot be used as an expression. This also means that the value returned by the Java method is discarded after the execution of the statement, and any data type is accepted as valid return type for the method, `void` included. Intuitively,

this is the main difference between `do native` statements and `native` expressions, because the methods invoked by `native` expressions are required to return a value of a Jadescript-compatible type.

A `native` expression can be used as part of any expression, e.g., as an operand of a binary operation. The syntax of a `native` expression, referred as *NExpr*, is the following:

⟨*NExpr*⟩ ::= 'native' ⟨*MetID*⟩ ('('⟨*ArgList*⟩')')? ('as' ⟨*Type*⟩)?

where *MetID* follows the same syntactic rules used in the `do native` statement, and similarly, *MetID* can be used to specify the resolution mode (compile-time versus runtime) of the method. For compile-time resolutions, the compiler checks, in addition to the conformance of the argument types to the parameter types of the resolved method, the compatibility between the Java return type of the method and the expected type of the expression. In this case, the `as` clause with the *Type* specification is optional. For runtime resolution, however, the compiler cannot infer the type of the expression because no information about the invoked method is provided at compile-time. In this case, the `as` clause is mandatory and the data type of the value returned by the method must conform to the type specified by *Type*, or an exception is thrown.

2.5 Native Ontology Concepts

Jadescript (communication) ontologies are abstractions provided to let programmers define a set of entities that are of primary importance for agent communication. As a matter of fact, ontologies can be used to ensure that agents share the same interpretation of the contents of exchanged messages. Agents can explicitly use one or more ontologies. When they use ontologies, they can create values of the entities declared in the used ontologies, manipulate those values, and use them as contents of messages. One of the main abstractions provided by ontologies are *concepts*. Concepts are entities with a structure defined in terms of *properties*, and they can be used as terms of other entities like other concepts, or, e.g., *predicates* to express logical facts about them, and *agent actions* to refer to actions that agents can perform.

The support for Jadescript-Java interoperability presented in this paper introduces a new type of ontology declaration, which is a different flavor of concept and it is called *native concept*. Native concepts are concepts whose concrete implementation is defined by the programmer in Java. To create a native concept, the Jadescript programmer needs to declare it as an ordinary concept in an ontology declaration, but prepending the `native` keyword to it. By doing this, the compiler generates a Java abstract class with the same name of the concept, and it includes a partial implementation of it. In particular, for each property declared in the concept, two abstract methods, namely a setter method and a getter method, are included.

When using native concepts, the programmer is supposed to write a concrete Java class that extends the generated abstract class with the additional requirement of including a constructor with no parameters in the concrete class. This is important because, when a concept is deserialized from the content of a message, its constructor with no parameters is invoked by the receiving agent before automatically populating the properties of the concept using the setter methods. After the concrete class is defined, the programmer can bind it to the ontology declaration by invoking the public static method named `bindNative` of the `Jadescript` class.

The adopted approach to support native concepts in Jadescript has several advantages. First, the programmer is guided in the definition of the concrete class by the Jadescript and Java compilers, ensuring that the programmer writes Java code that is usable from Jadescript. Second, each execution environment can have its own concrete implementation of a native concept, while, at the same time, preserving the details of the concept that are essential for message exchange. For example, consider a MAS composed of several agents executing in two connected containers, one on a desktop computer, the other on an Android smartphone. These agents can talk about abstract windows sharing a common interpretation of the graphical interface, which is only concerned with properties like its title and its graphical components. However, when an agent on the desktop chooses to construct a window visible to the user, a `JFrame` instance from the Swing library is shown. At the same time, when an agent on the Android smartphone chooses to show a window, an Android activity is started and activated.

2.6 Native Functions and Procedures

Jadescript agent and behaviour declarations include several event handlers, properties, procedures, and functions. In particular, these last two constructs can be used to define parameterized sections of procedural code (possibly, with side effects). The main differences among these two constructs is that functions are supposed to return a value, so they have a return type known at compile time, and they can be applied as part of expressions. Instead, procedures are supposed not to return a value, and they can be executed only by means of the `do` statement. Jadescript allows the declaration of top-level functions and procedures, which

```
module examples.math

native function sqrt(x as real) as real
```

Listing 3: Declaration of a native function.

ensures that they can be part of a module instead of being internal to a specific agent or behaviour declaration. Top-level functions and procedures can be used from anywhere inside the module, or they can be imported into other source files outside the module by using `import` declarations.

Top-level functions and procedures have been recently extended with the possibility of implementing them in Java, using an approach similar to native concepts described previously. Native function and procedure declarations are syntactically similar to ordinary top-level functions and procedures. The only differences are that they are introduced by the `native` keyword, and they have no body (and, consequently, no `do` keyword at the end of their header). For example, Listing 3 shows the declaration of a native function named `sqrt` that accepts one real parameter and computes one real value.

For each native function and procedure declaration, the Jadescript compiler generates a Java interface with the same name and a single Java abstract method. The abstract method has a parameter for each one of the parameters declared in the Jadescript native counterpart, and an additional first parameter of type `InvokerAgent` named `invokerAgent`. This reference provides a façade that allows performing operations on behalf of the agent, e.g., to write messages to the container message log, to shut the agent down, or to activate or deactivate behaviours. Actually, the availability of the `invokerAgent` argument is the one of the main advantages of using native functions and procedures instead of native expressions and statements. Note that an `InvokerAgent` instance is substantially different from the `JadescriptAgentController` instance described in Sect. 2. The former represents the agent from an internal perspective, i.e., it is the agent who called the native function or the native procedure, and it is used to perform operations within the Java thread of the agent. The latter, instead, is a proxy for the agent intended to control the agent from the outside, to issue commands, or to notify events from (possibly) another Java thread.

In order to use a native function or procedure, the programmer is supposed to create a concrete implementation of the generated interface. For example, Listing 4 shows a simple implementation of the native function declared in Listing 3. Note that the implementation class that provides the bodies of native

```
public class SqrtImpl
    implements examples.math.sqrt {
  public Double sqrt(InvokerAgent
    invokerAgent, Double x) {
    return Math.sqrt(x);
  }
}
```

Listing 4: Implementation of the native function `sqrt`.

functions and procedures must be bound at runtime by means of the method named `bindNative` available in the `Jadescript` class, similarly to the mechanism avaiable for native concepts.

3 RELATED WORK

Jadescript is not the first AOP language that provides support for interoperability with Java. As a matter of fact, several AOP languages are designed to work within Java-based environments. One language that shares relevant similarities with Jadescript, especially in the architecture of its main implementation, is SARL (Rodriguez et al., 2014; Feraud and Galland, 2017). SARL is an AOP language based on Xtend, which is a Java dialect that is tightly integrated with Xtext, which is a tool that is used by the current implementation of the Jadescript compiler. Thanks to this design choice, SARL code can readily use code defined in Java or in Xtend. The SARL programmer can easily access fields, invoke constructors, and invoke instance and static methods with the common dot notation. SARL, through its features inherited by Xtend, also provides OOP abstractions to directly define classes, interfaces, enumerations, and annotations. This approach allows writing code that is easily interoperable with Java code, and this is possible ultimately because SARL adopts both AOP and OOP paradigms. Jadescript was designed not to be an OOP language, and therefore it cannot enjoy the benefits of immediate interoperability with Java like SARL does.

Jason (Bordini and Hübner, 2006) is an AgentSpeak(L) (Rao, 1996) interpreter written in Java. AgentSpeak(L) provides a way to express agent programs based on the *Belief-Desire-Intention* (*BDI*) agent architecture, which equips agents with practical reasoning abilities. Jason extends AgentSpeak(L) with several additions, e.g., plan failure handling, and agent communication based on speech acts. Moreover, it implements the underlying BDI agent architecture in Java, offering programmers the possibility to customize the architecture by extending Java classes.

The interpreter loop is at the core of the Jason agent architecture. The loop is composed of steps to which parts of the reasoning process of the agent are delegated. Almost all parts of the loop can be customized (Bordini et al., 2007) by extending the `Agent` class and overriding the corresponding Java methods. Moreover, the environment the agents are situated in is implemented in Java using a specific API to implement environments and to provide sensing capabilities to agents. Finally, the body of Jason plans are sequences of private actions and of updates to the belief base or to the set of goals to be achieved by the agent. The list of actions that can be executed, and the effects of these actions, is defined by the environment or by internal actions, which are instances of Java classes that implement the `InternalAction` interface.

ASTRA (Dhaon and Collier, 2014; Collier et al., 2015) is an implementation of AgentSpeak(L) extended with teleo-reactive programming capabilities and support for encapsulated rules. ASTRA introduces a specific abstraction for interoperability with Java called module. A module is a direct mapping of a Java object into a namespace of agent actions and sensors. Each action and sensor corresponds to a Java method annotated with `@ACTION` and `@SENSOR` in the user-defined module class, which extends the ASTRA class `Module`. The module code can access and modify the internal state of the agent by means of the inherited field `agent`. For example, an action or a sensor implemented in Java can use `agent.beliefs().addBelief(b)` to update the belief base of the agent with the addition of a new belief `b`. At the beginning of each interpreter cycle, all the sensor methods of each used module are invoked to give them the opportunity to change the belief base or to emit appropriate events. Instead, actions are invoked explicitly within ASTRA plans by means of the ubiquitous dot notation. ASTRA modules are designed for reusability and their usage is expressed with statements in the body of an agent declaration.

An abstraction similar to Jason environments or ASTRA modules is absent in Jadescript because Jadescript provides no abstraction to explicitly model an environment. However, the sensing and acting abilities provided by them and by Jason internal actions can be easily defined in terms of the native statements, expressions, events, functions, and procedures discussed in Sec. 2.

4 CONCLUSION

Software agents provide a unique opportunity to promote reusability and adaptability of quality software (Bergenti and Huhns, 2004). However, this desirable result cannot be effectively achieved without a means to enable smooth interactions between agents and other pieces of software developed in using mainstream, or even legacy, technologies. For this reason, an AOP language that aspires to be used for the development of real-world agent-based solutions needs to provide an effective support for interoperability with mainstream languages like Java. This paper presented and discussed the main additions to Jadescript to support interoperability with Java, highlighting the importance of such additions and comparing their design with similar supports provided by other AOP languages, namely SARL, Jason, and ASTRA.

The set of discussed features can be further developed and improved in the future. For example, Jadescript does not currently provide a support to customize parts of the internal agent architecture, like Jason and several other AOP languages do. The prospect of this support is secondary to additional work that might be addressed in future versions of Jadescript. In particular, future work might address the definition of the subtyping and inheritance mechanisms to reuse agent and behaviour definitions. This improvement to the language does not seem particularly involved because Jadescript is designed to translate these definitions to Java classes. In addition, the Jadescript runtime support could provide the facilities to let programmers extend these classes to directly customize the architectural details of a class of agents or behaviours. This improvement of the language would allow the injection of custom code in the various phases of the agent and behaviour lifecycles. Moreover, it would provide the ability to change the default implementations of the core mechanisms of the agents, e.g., the behaviour scheduling algorithm or the algorithm that puts inbound messages into the private message inbox of the agent.

Finally, it is worth noting that additional discussions on interoperability features are expected if Jadescript would target other platforms than the Java virtual machine in the future. The interoperability features presented in this paper are general enough to easily support such a prospect, ensuring that the syntax and the semantics of the interoperability features would remain the same from the Jadescript side, independently of the target platform. However, other languages and platforms could inspire new ideas for interoperability features or present unforeseen requirements and constraints.

ACKNOWLEDGEMENTS

This work was partially supported by the Italian Ministry of University and Research under the PRIN 2020 grant 2020TL3X8X for the project *Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems* (T-LADIES).

REFERENCES

Bellifemine, F., Bergenti, F., Caire, G., and Poggi, A. (2005). JADE–A Java Agent DEvelopment Framework. In *Multi-Agent Programming*, volume 25 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 125–147. Springer.

Bellifemine, F., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*. Wiley Series in Agent Technology. John Wiley & Sons.

Bergenti, F., Caire, G., and Gotta, D. (2015). Large-scale network and service management with WANTS. In *Industrial Agents: Emerging Applications of Software Agents in Industry*, pages 231–246. Elsevier.

Bergenti, F., Caire, G., Monica, S., and Poggi, A. (2020). The first twenty years of agent-based software development with JADE. *Autonomous Agents and Multi-Agent Systems*, 34(36).

Bergenti, F. and Huhns, M. N. (2004). On the use of agents as components of software systems. In *Methodologies and Software Engineering for Agent Systems*, pages 19–31. Springer.

Bergenti, F., Monica, S., and Petrosino, G. (2018). A scripting language for practical agent-oriented programming. In *Proceedings of the* 8th *ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018) at ACM SIGPLAN Conference Systems, Programming, Languages and Applications: Software for Humanity (SPLASH 2018)*, pages 62–71. ACM.

Bergenti, F. and Petrosino, G. (2018). Overview of a scripting language for JADE-based multi-agent systems. In *Proceedings of the* 19th *Workshop "From Objects to Agents" (WOA 2018)*, volume 2215 of *CEUR Workshop Proceedings*, pages 57–62. RWTH Aachen.

Bordini, R. H. and Hübner, J. F. (2006). BDI agent programming in AgentSpeak using Jason. In *Proceedings of the* 6*th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA 2005)*, volume 3900 of *Lecture Notes in Artificial Intelligence*. Springer.

Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. Wiley Series in Agent Technology. John Wiley & Sons.

Caire, G., Bellifemine, F., Trucco, T., and Rimassa, G. (2010). *Jade Administrator's Guide*. Available at jade.tilab.com.

Collier, R. W., Russell, S., and Lillis, D. (2015). Reflecting on agent programming with AgentSpeak(L). In *Lecture Notes in Computer Science*, volume 9387. Springer.

Dhaon, A. and Collier, R. (2014). Multiple inheritance in AgentSpeak(L)-style programming languages. In *Proceedings of the* 4th *ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2014) at ACM SIGPLAN Conference Systems, Programming, Languages and Applications: Software for Humanity (SPLASH 2014)*.

Feraud, M. and Galland, S. (2017). First comparison of SARL to other agent-programming languages and frameworks. In *Proceedings of the* 8*th International Conference on Ambient Systems, Networks and Technologies (ANT 2017) and of the* 7*th International Conference on Sustainable Energy Information Technology (SEIT 2017)*, volume 109 of *Procedia Computer Science*. Elsevier.

Genesereth, M. R. and Ketchpel, S. P. (1994). Software agents. *Communications of the ACM*, 37(7):48–55.

Iotti, E., Petrosino, G., Monica, S., and Bergenti, F. (2020). Exploratory experiments on programming autonomous robots in Jadescript. In *Proceedings of the* 1*st Workshop on Agents and Robots for Reliable Engineered Autonomy (AREA 2020) at the European Conference on Artificial Intelligence (ECAI 2020)*, volume 319 of *Electronic Proceedings in Theoretical Computer Science*. Open Publishing Association.

Petrosino, G. and Bergenti, F. (2019). Extending message handlers with pattern matching in the Jadescript programming language. In *Proceedings of the* 20th *Workshop "From Objects to Agents" (WOA 2019)*, volume 2404 of *CEUR Workshop Proceedings*, pages 113–118. RWTH Aachen.

Petrosino, G., Iotti, E., Monica, S., and Bergenti, F. (2021). Prototypes of productivity tools for the Jadescript programming language. In *Proceedings of the* 22nd *Workshop "From Objects to Agents" (WOA 2021)*, volume 2963 of *CEUR Workshop Proceedings*, pages 14–28. RWTH Aachen.

Petrosino, G., Iotti, E., Monica, S., and Bergenti, F. (2022a). A description of the Jadescript type system. In *Proceedings of the* 3*rd International Conference on Distributed Artificial Intelligence (DAI 2022)*, volume 13170 of *Lecture Notes in Computer Science*, pages 206–220. Springer.

Petrosino, G., Monica, S., and Bergenti, F. (2022b). Robust software agents with the jadescript programming language. In *Proceedings of the* 23rd *Workshop "From Objects to Agents" (WOA 2022)*, CEUR Workshop Proceedings. RWTH Aachen.

Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In *MAAMAW 1996: Agents Breaking Away*, pages 42–55. Springer.

Rodriguez, S., Gaud, N., and Galland, S. (2014). SARL: A general-purpose agent-oriented programming language. In *Proceedings of the IEEE/WIC/ACM International Joint Conferences of Web Intelligence (WI 2014) and Intelligent Agent Technologies (IAT 2014)*, volume 3, pages 103–110. IEEE.