Prototypes of Productivity Tools for the Jadescript Programming Language

Giuseppe Petrosino[1], Eleonora Iotti[1], Stefania Monica[2] and Federico Bergenti[1]

[1]Dipartimento di Scienze Matematiche, Fisiche e Informatiche, Università degli Studi di Parma, Italy
[2]Dipartimento di Scienze e Metodi dell'Ingegneria, Università degli Studi di Modena e Reggio Emilia, Italy

Abstract
Jadescript is an agent-oriented programming language built on top of JADE. So far, the focus of the development of the language was on design choices, on syntax refinements, and on the introduction of expressions and constructs for agent-related abstractions and tasks. In this paper, a proposal to achieve the crucial goal of making Jadescript suitable for professional use is presented. The success of Jadescript, as a solid language to build real-world agent-based software systems, is necessarily related to its effective integration with mainstream development tools. In this paper, some of the productivity tools developed to integrate Jadescript with a mainstream development environment are presented as a way to promote the successful adoption of the language towards the community of JADE users.

Keywords
Agent-Oriented Programming Languages, Agent-Oriented Software Engineering, Jadescript, JADE

1. Introduction

The search for novel and effective development technologies to design agents and to build multi-agent systems is a fast-growing research issue. Over the years, agents were employed in many different application scenarios, as illustrated, e.g., in [1], where the relevant impact of agent technologies is discussed. Example application scenarios are agent-based simulations [2], distributed constraints reasoning [3], accurate indoor positioning [4, 5, 6, 7, 8, 9, 10], serious games [11, 12, 13], and network management [14], just to cite some.

In general, multi-agent systems are key tools for those problems when goals to be achieved are sufficiently complex to require the coordination and cooperation of a large number of different agents, often distributed on many hosts, that must use their skills in an effective way to achieve a collective goals. Multi-agent systems open to a plethora of new design and development problems, such as agent communication and interaction protocols, message passing and routing, deployment of agents to network hosts, the reception of shared environment information, norms and validations, and so on. Some of such interesting problems are taken over by *AOSE* (*Agent Oriented Software Engineering*) [15] researchers to produce a remarkable range of solutions and frameworks, as well as sophisticated software platforms [16].

WOA 2021: Workshop "From Objects to Agents", September 1–3, 2021, Bologna, Italy
giuseppe.petrosino@studenti.unipr.it (G. Petrosino); eleonora.iotti@unipr.it (E. Iotti); stefania.monica@unimore.it (S. Monica); federico.bergenti@unipr.it (F. Bergenti)
0000-0001-6234-5328 (G. Petrosino); 0000-0001-7670-2226 (E. Iotti); 0000-0001-6254-4765 (S. Monica); 0000-0002-4756-4765 (F. Bergenti)

Most, if not all, AOSE methods and tools target the *AOP* (*Agent-Oriented Programming*) [17, 18] paradigm, which explicitly treat the concept of agent as a basic building block, ready to be used by the programmer. Nonetheless, many AOP frameworks and platforms provide software libraries written in some *GPL* (*General Purpose Language*), and such libraries extend the usage of the chosen language to AOP problems. As a matter of fact, a common way to enrich a GPL with custom functionality, e.g. agent-based functionality, is to provide specific *APIs* (*Application Programming Interfaces*). Alternatives to the use of GPLs in software development are various, and its worth citing at least two of such possibilities, namely, *DSLs* (*Domain Specific Languages*) and scripting languages.

These two alternatives, revisited in AOSE, open to a relevant advancement in the direction of AOP, which is the use of *APLs* (*Agent Programming Languages*) [19]. Such languages not only support agent-based features, but they also put them on a language level. Jadescript [20, 21, 22], a language built on top of *JADE* (*Java Agent DEvelopment framework*) [23, 24], is an example of an AOP scripting language. Jadescript is a significant extension of a previous APL called JADEL [25, 26, 27, 28, 29, 30, 31] that incorporates the features of an AOP scripting language. Other popular APLs are *Jason* [32], which is an implementation of *AgentSpeak(L)* [33], *3APL* [34], *GOAL* [35], *SARL* [36], and several others [37].

It is common opinion that the major benefit of adopting a specific language for agent development is the availability of native abstractions, constructs, and expressions in the language to explicitly recall the agent domain, putting agents as first-class citizens of the language. On the other hand, despite the great results in terms of effectiveness and usability, most of the aforementioned APLs has a niche user base, composed mainly by researcher and students. Pure agent-based programming seems yet relegated to academic environments, despite many real-world applications use multi-agent systems on a daily basis. A reason for that, among many others, lies in the preferences of programmers and in programming trends. A successful language for a wider audience must take into account such preferences and trends, making its idioms as simple and readable as possible, yet not ambiguous. Unfortunately, these design choices alone are not sufficient to bring success to programming languages. The core functionalities offered by a language are appreciated when they are stable and reliable, making the language usable for robust applications. Therefore, the proposal of a novel APL should not only regard the accurate design and implementation of desired functionality by means of syntactical categories and their semantics, but also the construction of an adequate ecosystem. Such an ecosystem could be defined as the set of all tools, utilities, and interfaces that help programmers in their daily coding routine, i.e., those services offered by *IDEs* (*Integrated Development Environments*), frameworks, libraries, and related tools.

In this paper, the main steps taken to build such tools, utilities, and interfaces for the Jadescript language are described. The adopted approach aims at making the language completely integrated with the Eclipse IDE [38] and its plugins, thus providing tools such as a Jadescript perspective, some specific Eclipse wizards, a dedicated syntax highlighting editor, and a launch system for agents and agent containers. This work is primarily based on another Eclipse plugin, called Xtext, which generates some ready and easy-to-use tools for DSLs. Such tools were then adapted to the agent domain and made suitable for the user experience envisaged for Jadescript programmers. The resulting ecosystem is an important step for the growth of Jadescript, and it brings to the language the professional feeling that JADE users expect.

This paper is structured as follows. First, in Section 2, an introduction to Xtext and related technologies is provided to give the reader sufficient background information on how to build a professional tool for the Eclipse IDE using Xtext. Then, in Section 3, the core features of the Eclipse plugin for Jadescript are detailed, taking into account the Xtext extensions and the Eclipse extensions. Finally, a discussion on the main results of the approach adopted in the development of presented tools is provided to conclude the paper.

2. Overview of Xtext

Xtext [39] is the main software used to create the presented tools related to Jadescript. Xtext is an open-source framework for the development of DSLs and programming languages. It is designed to lift most of the burden of the programming language designer, not only by taking the usual tasks of a parser generator, but also by providing a set of advanced tools that guide in the construction of a complete compiler and a full-featured IDE.

An Xtext language project is made of several related Eclipse projects. Three of them are the most important:

1. The main project, which contains the grammar, the support for the semantics, and all the other components for the language that are independent from the *UI* (*User Interface*);
2. The IDE project, which contains the code for the general behaviour of the UI, regardless of the specific target environment, so the code in this project can be specialised, for example, for an IntelliJ IDEA [40] plugin, or for an editor embedded in a Web page; and
3. The UI project, which depends on the IDE project and contains the specific details related to the Eclipse UI to implement a custom language plugin for the Eclipse IDE.

The main project contains the entry point for the language design process, which is the Xtext grammar file for the language. Xtext is grammar driven and it provides a grammar language to generate, from a single source file, all the essential elements of the skeleton of the compiler and of the other tools.

The first essential component generated by Xtext is the lexer, which is generated from the terminals defined the grammar. The lexer is usually complete and sufficient in most of the cases. However, it is worth mentioning that the default behaviour of the lexer was specialised for Jadescript because Jadescript is a language with semantically relevant indentation. Section 3 describes the details of this specialisation.

The second component generated by Xtext is the parser. The parser is used by the compiler and by the editor to obtain an *AST* (*Abstract Syntax Tree*) from a processed text. Since the Xtext grammar language is an extension of the grammar language of the ANTLR [41] parser generator, the generated parser employs a $LL(*)$ [42] parsing strategy.

Together with the parser, Xtext generates a syntax validator. This component is in charge of isolating the portions of a processed text that are syntactically incorrect to produce the corresponding errors, warnings, and recommendations for the user.

Finally, Xtext produces a metamodel of the generated language using *EMF* (*Eclipse Modeling Framework*) and its metamodel format, called Ecore. A set of Java interfaces and classes is created to represent an object model of the grammar. For each non-terminal grammar rule, a

Java `EObject` class is generated, where each component of the rule is mapped to one of the properties of the class. This provides the language designer with a statically typed programming interface to work with the ASTs generated by the parser. Note that an important difference between an Xtext grammar and an ANTLR grammar is the possibility to add additional metadata to customise the aspects of the generation of the object model of the language. Another notable difference between the two grammar languages is the possibility to inject, directly in the model of the generated AST, useful metadata like, for example, the type of syntax element that a reference can be linked to. Such a metadata comes in handy when working with the AST in the portions of the compiler that define the semantics of the language.

With these essential components automatically generated by Xtext from the grammar file, the language designer has already a working editor and other working tools for code editing and syntactical validation. However, to actually interpret and generate executable code, it is required to add custom components. This is achieved by a widespread adoption of the *DI* (*Dependency Injection*) design pattern. The main idea of such an approach is that all the classes that implement the functionalities provided by the final plugin refer to their main dependencies by declaring them as fields annotated with a dedicated annotation. The Xtext framework uses a class to define the bindings among the Java interfaces of these dependencies with their respective implementations. Such bindings are then used by an injector object to create all the components and their dependencies at runtime. The module class is open for extensions and all the binding methods can be overridden to provide the language designer with the ability to change, in a controlled and structured way, almost any aspect the generated tools.

Among such customisable aspects, two require particular attention, namely code validation and code generation. Semantic validation of the code is managed by a validator class. In Xtext, such a class is implemented with a declarative approach. The language designer specifies, by adding methods to the class, which types of the Java object model of the AST are to be checked. In case of erroneous or problematic code, such methods can build and report errors, warnings, and recommendations that are used as feedback to the user.

Code generation can be achieved by means of a class that extends the `IGenerator` interface, which defines how to generate new texts starting from the AST obtained from the parsing of a source file. For languages targeting the *JVM* (*Java Virtual Machine*), however, another specialised approach is available, based on the `JVMModelInferrer` class. By extending this class, the language designer is able to symbolically declare which Java classes, interfaces, methods, and fields are generated from each source file. Xtext keeps track of such mappings and use them to ease the implementation of the language tools by partially generating:

1. A type checking system, based on the Java type system;
2. A scope provider, which is able to resolve references to Java packages, types, and symbols;
3. A code generator, which creates the Java source files corresponding to the declared Java structures provided by the `JVMModelInferrer` object; and
4. Some IDE features like basic auto-completion support, symbolical navigation in the editor, linking between written and generated code, and basic refactoring.

Obviously, if the validator or the type checker find errors in the source code, the compiler aborts code generation, signalling the problems to the user.

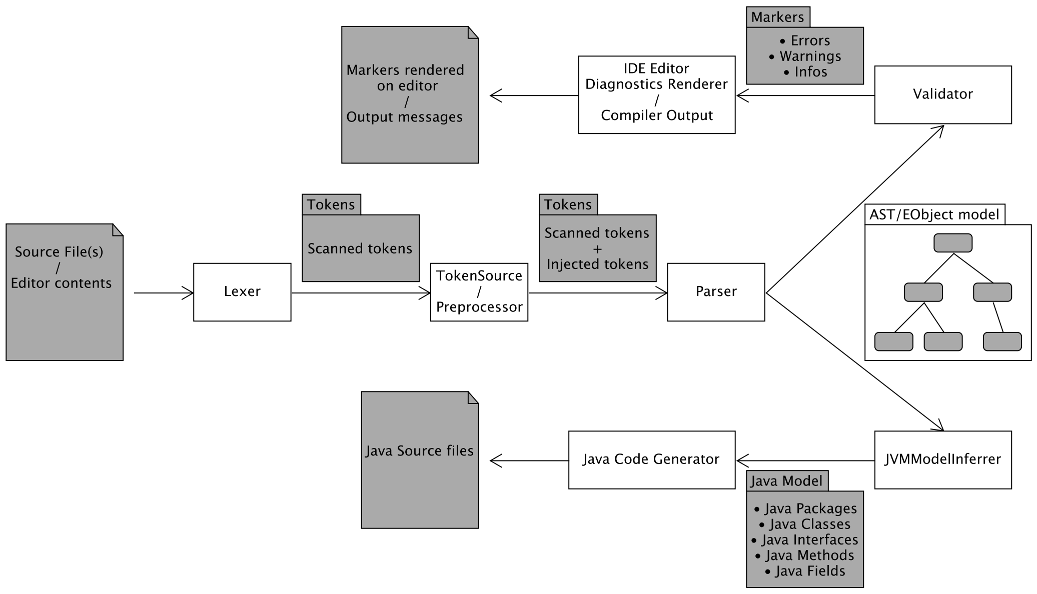


Figure 1: The compilation and validation processes in the Xtext framework for languages that target the Java virtual machine.

Jadescript is currently a language based on the JVM. More specifically, it is compiled to Java code. For this reason, the Jadescript compiler uses the `JVMModelInferrer` class to take advantage of the prebuilt mechanisms generated by Xtext for JVM-based languages. The general outline of the validation and code generation processes provided by Xtext and used by the Jadescript compiler is schematised in Fig. 1.

When the user feeds the compiler with a set of Jadescript source files, the Xtext runtime provides the contents of the files to the lexer generated from the Jadescript grammar. The lexer produces a stream of tokens, and the `TokenSource` interface, provided by Xtext, is used to preprocess the stream of tokens to inject into the stream needed synthesised tokens relative to the semantically relevant indentation. Actually, the original stream of tokens is analysed to identify the points in the stream where the level of indentation changes. Synthesised tokens are injected at the identified points in the stream to ensure that changes in the indentation are properly reported to the parser. The new stream of tokens is fed to the parser, which produces an AST and an EMF model of it. These results are then fed to the validator, which statically analyses the code in search for problems, following the semantic rules of the language. If the validator does not find any errors in the code, the same AST is reused and provided to the `JVMModelInferrer` class, which produces an intermediate representation, namely the Java model of the target code. This representation is finally used by the framework to produce the Java code that Eclipse eventually compiles for the JVM.

3. The Jadescript Eclipse IDE Plugin

The discussed Jadescript development tools include an Eclipse IDE plugin, which contains all the software tools to write Jadescript code, to create and manage projects, to create and edit source files, and to launch and debug agents. The plugin was created with the help of Xtext, especially for those features of it that are strongly related with the syntax and the semantics of the language, and that, therefore, are part of the Jadescript compiler. However, some of the tools, as discussed in this section, were created by means of the tools provided by the Eclipse *PDE* (*Plugin Development Environment*).

3.1. Xtext Extensions

As mentioned in Section 2, in Xtext, new language projects enjoy of a set of interesting features implemented by default and generated automatically from the grammar file. However, some aspects of the generated code require the language developer to adapt the default Xtext implementations by overriding specific methods with custom methods. For Jadescript, two aspects required specialised implementations, namely, the `TokenSource` implementation used to manage semantically relevant indentation, and the set of mechanisms that implement the semantics of the language in the compiler.

3.1.1. Token Source System

In an Xtext-generated parser, the parser reads the input tokens from a `TokenSource` object, which is an object that produces tokens from source code upon request from the parser. In most programming languages, the parser assumes that whitespace characters (i.e., blanks, tabulations, newline characters) are hidden in the grammar and not considered in the input stream of tokens. The purpose of whitespace characters is to act as separators among parts of the text that are relevant for the grammar. This is not completely true for languages with semantically relevant indentation like Jadescript. In such languages, the level of indentation of a line not only keeps the code tidy and easy to read, but it is also used by the compiler to understand where the line is placed in the structure of the code. For example, for procedural code, some statements are expected to include inner blocks of code (e.g., the *then* branch of an *if* statement, or the body of a *loop* statement), and the lines belonging to such blocks of code have to start with an inner level of indentation. At the same time, in Jadescript and other modern programming languages, the sequential composition of statements is not expressed by an explicit *end-of-statement* operator symbol (e.g., `;` for C-like languages). Such separation of statements has to be inferred by the compiler using the newline character as hint. In this inference mechanism, the compiler has to leave the possibility for the user to split any statement in two or more lines, whenever such statements are too long and the programmer wishes to increase readability.

In Jadescript, this set of behaviours is encoded in a special kind of tokens (also known as synthetic tokens) that signals the parser of three types of relevant points in the code:

1. The point of termination of a statement (*NEWLINE* synthetic token);
2. The point where a new code block is opened (*INDENT* synthetic token); and
3. The point where a previously opened code block is closed (*DEDENT* synthetic token).

The `TokenSource` interface of Xtext is then implemented by the `JadescriptTokenSource` class that includes an algorithm that injects the synthetic tokens mentioned above according to a simple set of rules.

By default, when the parser requests a new token, the `JadescriptTokenSource` object simply responds with the next token that the lexer generated by scanning the source code. However, when at least one newline characters is encountered in the stream, the `JadescriptTokenSource` object computes the indentation level of the new line. If the indentation is more in depth than the previous line, and the previous line ends with `do` (keyword that, in Jadescript, is used in many constructs to express the beginning of the definition of a procedural body) or the new line starts with any of the following keywords {`concept`, `proposition`, `predicate`, `action`, `function`, `procedure`, `on`, `property`, `execute`}, then an *INDENT* synthetic token is injected in the stream. When the indentation is more in depth than the previous line, but those keywords are not present, the `JadescriptTokenSource` object does not inject any new token in the stream. This last rule allows users to split a line into two, indenting the second line, to simply improve readability without changing the semantics of the code in the lines. If, however, the indentation level is the same as the previous line, a *NEWLINE* token is injected, signalling the parser that a statement (or declaration) ended with the previous line and that a new statement (or declaration) starts with the new line. Finally, when the indentation is less in depth than the previous line, a number of *DEDENT* tokens are injected corresponding to the number of blocks being closed.

3.1.2. Jadescript Semantic Classes

After parsing, the compiler created by the Xtext framework produces a Java object model of the AST. This can be navigated to perform the computations required by the semantics of the language. These compiler computations, in the Jadescript compiler, are handled by a set of Java classes called semantic classes. Each one of these classes handles how a particular node of the AST is used for code generation and validation.

The semantic classes can be subdivided in four categories, each one referring to a type of construct of the language. The following paragraphs describe these categories, sorted by structural depth level.

Top Level Entities. These semantic classes implement the semantics of those top-level declarations (e.g., *agent*, *behaviour*, and *ontology*) that can be written directly inside a file, not contained in any other construct or declaration. When the compiler walks the AST on these types of nodes, they are mapped directly to JVM types (classes and interfaces) by means of the utilities provided by the `JVMModelInferrer` class generated by Xtext.

Entity Features. These elements of the language are the main building blocks of each top-level declaration. They are usually directly enumerated in the body of the declaration, and for this reason, they appear as indented by just one level in the source code. Examples of features include *event handlers* and *properties* in *agent* and *behaviour* declarations, and *concept* and *predicate* entries in *ontology* declarations. Entity features are usually compiled to Xtext-compatible JVM model elements, namely Java fields, methods and inner classes.

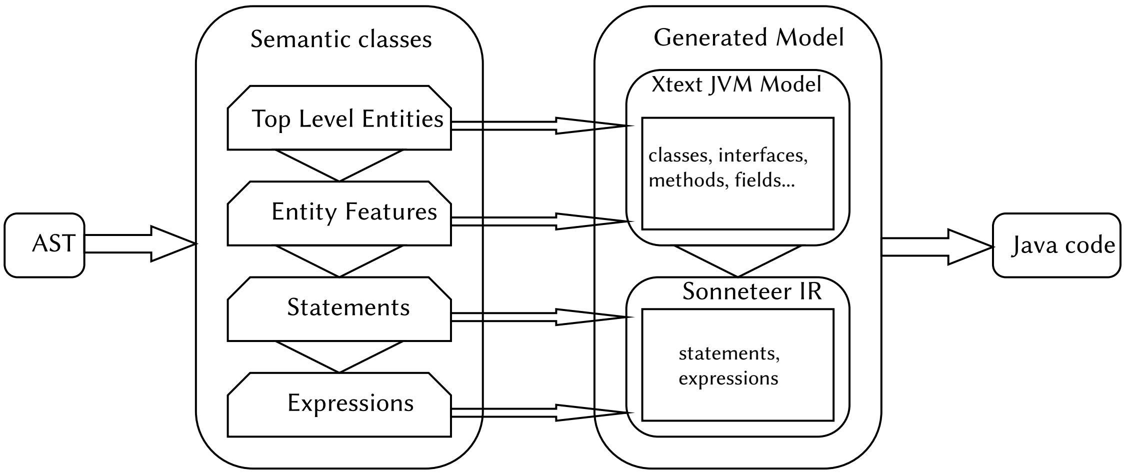
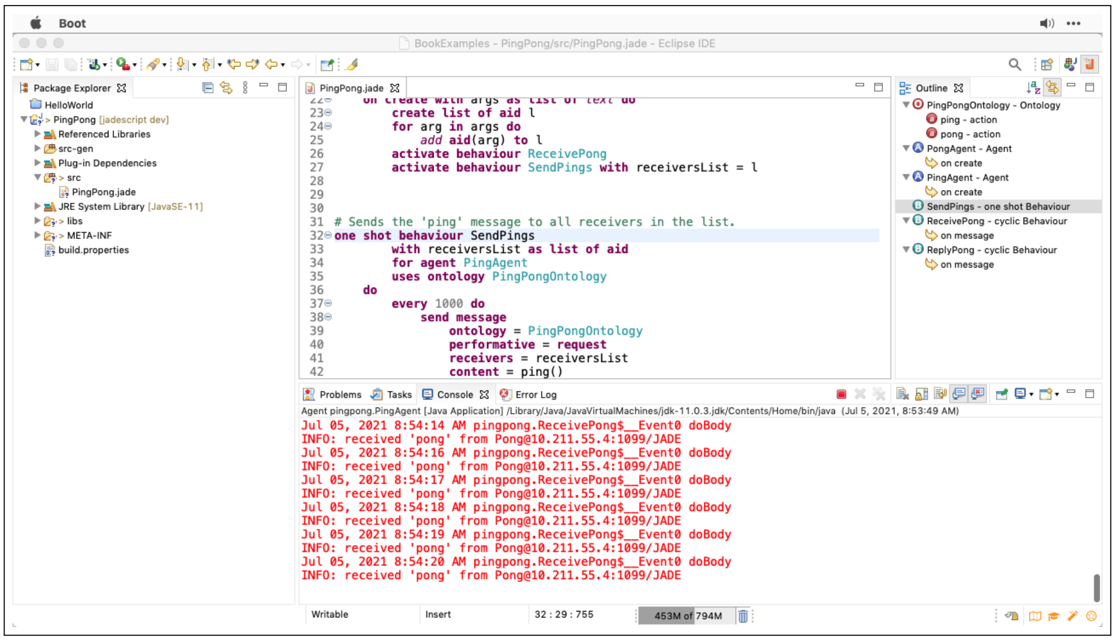


Figure 2: The categories of semantic classes and how their generated artefacts are composed.

Statements. Statements are the building blocks of the procedural portions of code. They are used in those entity features that require a procedural body, i.e., structured lists of commands. Note that such commands can include expressions or other procedural bodies (e.g., the guard and the body of a `while` statement). Statements are compiled by semantic classes into objects of a custom *IR* (*Internal Representation*) model, named Sonneteer. Sonneteer is a small Java library which implements a simple API to generate text strings of Java source code. The usage of this library in the implementation of the semantic classes ensured a good degree of type safety in the generation of structured Java code. Objects built with this library are used by the compiler in the code generation phase at the end of compilation, to compute the actual text content of the generated Java source code. The `send message` statement and the `activate behaviour` statement are good examples of elements of this category.

Expressions. As in many modern programming languages, the most fine-grained category of language constructs is expressions. Expressions are designed to be composable, and, in statically and strongly typed languages like Jadescript, each expression has a type, which is used by the compiler to check if some combinations of operations and operands is consistent with the rules of the language semantics. In Jadescript, the type of an expression is computed at compile time not only for validation purposes, but also to infer the type of variables and of agent/behaviour properties. Semantic classes related to this category compile expressions into simple text strings in three steps. In the first step, all the statements and expressions in a procedural code block are translated into Sonneteer objects, which include placeholder elements that annotate the generated code with compiler metadata. These placeholder elements are then analysed in the second step. The result of the analysis is finally used in the third step to perform optimisations. Note that expressions include literals, infix and unary operations like addition and logical negation, and other special operations like `matches` for pattern matching.



```
    on create with args as list of text do
        create list of aid l
        for arg in args do
            add aid(arg) to l
        activate behaviour ReceivePong
        activate behaviour SendPings with receiversList = l



# Sends the 'ping' message to all receivers in the list.
one shot behaviour SendPings
        with receiversList as list of aid
        for agent PingAgent
        uses ontology PingPongOntology
    do
        every 1000 do
            send message
                ontology = PingPongOntology
                performative = request
                receivers = receiversList
                content = ping()
```

```
Agent pingpong.PingAgent [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.0.3.jdk/Contents/Home/bin/java  (Jul 5, 2021, 8:53:49 AM)
Jul 05, 2021 8:54:14 AM pingpong.ReceivePong$__Event0 doBody
INFO: received 'pong' from Pong@10.211.55.4:1099/JADE
Jul 05, 2021 8:54:16 AM pingpong.ReceivePong$__Event0 doBody
INFO: received 'pong' from Pong@10.211.55.4:1099/JADE
Jul 05, 2021 8:54:17 AM pingpong.ReceivePong$__Event0 doBody
INFO: received 'pong' from Pong@10.211.55.4:1099/JADE
Jul 05, 2021 8:54:18 AM pingpong.ReceivePong$__Event0 doBody
INFO: received 'pong' from Pong@10.211.55.4:1099/JADE
Jul 05, 2021 8:54:19 AM pingpong.ReceivePong$__Event0 doBody
INFO: received 'pong' from Pong@10.211.55.4:1099/JADE
Jul 05, 2021 8:54:20 AM pingpong.ReceivePong$__Event0 doBody
INFO: received 'pong' from Pong@10.211.55.4:1099/JADE
```

Figure 3: A screenshot of the Jadescript perspective hosted in the Eclipse IDE.

3.2. Eclipse Extensions

Part of the Jadescript Eclipse plugin is implemented using directly the PDE. The Eclipse IDE is designed as an extensible framework, with facilities that ease the addition and customisation of functionality. Such customisations can be created in plug-ins, and the extensibility approach allows plug-ins to use and extend other plug-ins declaring a set of hooks in the extension points in the manifest XML file of the plugin. The Jadescript plugin uses these extension points to customise some aspects of the user interface of the IDE. The main extensions of the Eclipse IDE provided by the plugin are the Jadescript Perspective, the Wizards, the Syntax Highlighting in the Jadescript editor, and the Container and Agent launcher actions.

3.2.1. Jadescript Perspective

Fig. 3 shows a screenshot of the Jadescript Perspective as provided by the plugin. On the left, there is the *Package Explorer* View. It is a tree view of the current eclipse workspace and the contents of its projects. The contextual menu on these elements contains a set of common Eclipse project management actions, like importing and exporting, and operations to manage the view itself, like *Refresh*. However, it is enriched of several actions specific for the management of Jadescript projects, namely actions to start wizards for the creation of Jadescript projects and files (see section 3.2.2) and for running the Jadescript agents declared in the source files (see section 3.2.4). At the centre, there is the editor section. This acts as a container for editor tabs, including instances of the Jadescript Editor for the editing of Jadescript source files. On the right, the Jadescript perspective lays out the *Outline* View. This view shows the structure of the code of the currently focused file in a tree view. In this view the root nodes of the tree represent the top level declarations in the file, and their children represent their declared features, i.e., properties, event handlers, functions and procedures for agent and behaviour declarations, and concepts, predicates, propositions and actions for ontology declarations. The Outline View enjoys of a bidirectional linking between the contents of the file and the nodes, which is updated and rendered in real time.

3.2.2. Eclipse Wizards

The plugins includes a set of wizards for the creation of projects and source files. The *New Jadescript Project* wizard guides the user in the creation of a new project with the Jadescript nature. Jadescript projects always include three folders, created by the wizard. The `src` directory is where all Jadescript and Java source files written by the user should go. Jadescript files saved in this directory (and in its subdirectories) are used as input for the Jadescript compiler, which generates the corresponding Java files into the second directory, named `src-gen`. Finally, the `libs` directory contains a set of *JAR* (*Java ARchive*) libraries used by the project. The *New Jadescript Project* wizard always puts three JARs in this directory, which are the `jadescript.jar` file, which includes some required code for Jadescript (like the implementation of the base Jadescript *Agent* and *Behaviour* types), and the `jade.jar` and the Apache Commons Codec libraries, required for running JADE and Jadescript agents and platforms.

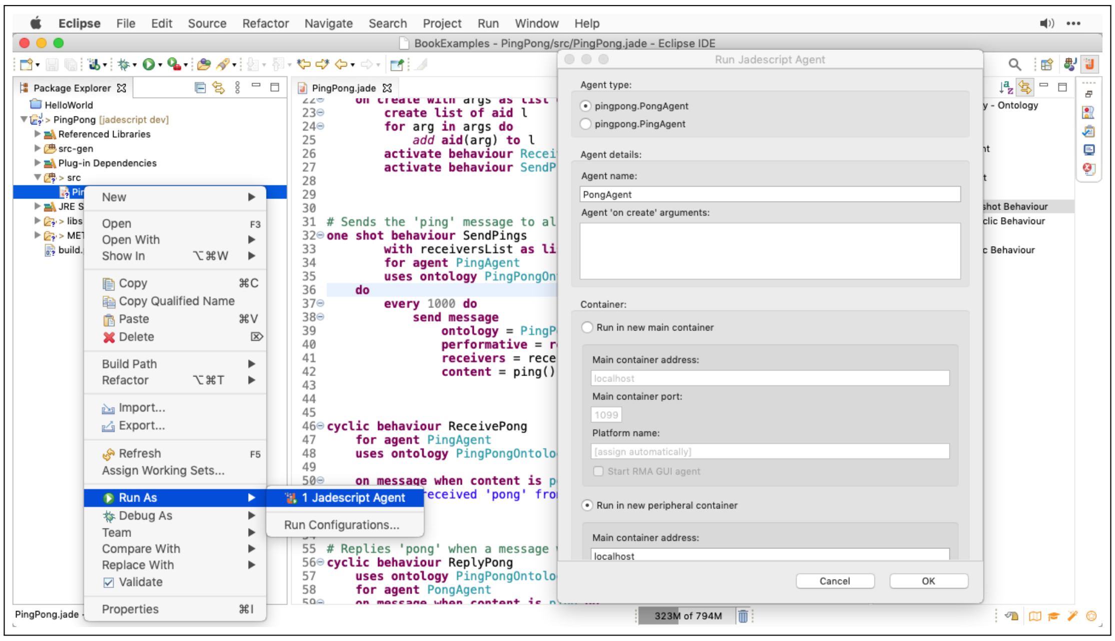
Four more wizards are used to guide the user to create new Jadescript source files. The *New Jadescript File* wizard creates a new empty Jadescript file in the specified project location, with the specified module. This wizard is the specialised into the *New Jadescript Agent*, *New Jadescript Ontology* and *New Jadescript Behaviour* wizards, which collect information from the user in order to create new Jadescript source files with the stubs of, respectively, an agent declaration, an ontology declaration, and a behaviour declaration.

3.2.3. Syntax Highlighting

As many modern programming editors, code is highlighted with different colours, in order to help the user quickly recognise and tell the various elements of the code. This aspect of the appearance of code in the Jadescript editor can be customised by the user via the Jadescript section of the Eclipse preferences, at the *Syntax Coloring* preference page. The syntax highlighting for Jadescript is advanced enough to make use of complex semantic rules to highlight the text of different colours, and this is done by using a set of special methods in the semantic classes. This approach allows, for example, to highlight with different colours the first assignment of a variable and its subsequent usages and re-assignments.

3.2.4. Container and Agent Launchers

Two fundamental entries in the extension points of the plugin implement two actions in the IDE. The first is accessible from the Eclipse toolbar, and it is used to launch a new JADE main container. By pressing this button, a new instance of the JADE Main container is launched locally on the machine where Eclipse is running. This action creates a new local agent platform, and it can be used as a starting point to build a complex network of JADE containers. The



```
    on create with args as list
        create list of aid l
        for arg in args do
            add aid(arg) to l
        activate behaviour Recei
        activate behaviour SendP



# Sends the 'ping' message to al
one shot behaviour SendPings
        with receiversList as li
        for agent PingAgent
        uses ontology PingPongOn
    do
        every 1000 do
            send message
                ontology = PingP
                performative = r
                receivers = rece
                content = ping()



cyclic behaviour ReceivePong
    for agent PingAgent
    uses ontology PingPongOntolo

    on message when content is p
        eceived 'pong' fro

# Replies 'pong' when a message
cyclic behaviour ReplyPong
    uses ontology PingPongOntolo
    for agent PongAgent
```

Agent type:
[✓ pingpong.PongAgent]
pingpong.PingAgent

Agent details:
Agent name: PongAgent
Agent 'on create' arguments:

Container:
Run in new main container
Main container address: localhost
Main container port: 1099
Platform name: [assign automatically]
Start RMA GUI agent
[✓ Run in new peripheral container]
Main container address: localhost

Figure 4: A screenshot of the agent launcher dialog for the Jadescript editor in the Eclipse IDE.

button is a pull-down button, with a second option available in its drop-down menu. By clicking on the second option, the created JADE main container includes a *RMA* (*Remote Monitoring Agent*) with a *GUI* (*Graphical User Interface*) that allows the developer to see the status of the platform and to create new containers and agents. The JADE software is launched using the Java classpath of the currently open project. In this way, from within the RMA GUI it is possible to launch new agents in the platform, using the Java classes generated from the Jadescript code.

The second action is accessible from the *Run As* submenu of the contextual menu. It is only accessible when a Jadescript source file is selected or open in the editor. When clicked, a dialog window opens. This window provides the user with the ability to launch a new agent in a new container. It itemises a list of selectable agent types, which correspond to the ones declared in the source file. The dialog then allows the user to customise various details of the agent, like its name, its input arguments, and the details of the container in which it will be created in.

4. Conclusions

In this paper, some of the productivity tools explicitly designed for Jadescript are presented and discussed. The approach that underlies such tools is driven by the goal of promoting Jadescript towards a professional ecosystem by considering the preferences of professional programmers together with the well-known practical advantages of modern IDEs. The target IDE for the current implementation of the discussed productivity tools is the Eclipse IDE, which is a well-known and appreciated tool that served Java and JADE programmers for several years. However, the porting of the tools to other popular IDEs has already been considered.

Xtext was used as the basic building block of the internals of the Jadescript compiler. Therefore, this paper provides a brief overview of Xtext to detail its functionality and to explain the relevance of such a tool for Jadescript. Actually, Xtext is not only used to generate Java code from Jadescript code, but it is also used for the validation of Jadescript code and for the generation of errors, warnings, and recommendations for the programmer.

After the brief description of Xtext, the paper discusses the developed Eclipse IDE Plugin for Jadescript starting with a short digression on the token source system and its Xtext extension. Such an extension is of primary importance for the tools presented in this paper because it provides the basic support for semantically relevant indentation. Note that semantically relevant indentation plays an important role to greatly improve readability of Jadescript codes and to give a modern appeal to the language. The presented Eclipse IDE Plugin allows programmers to take advantage of all the features that the Eclipse IDE provides for other languages like Java. In particular, the Jadescript perspective provides a customised view of the Eclipse IDE specifically designed to accommodate the needs of Jadescript programmers. The Jadescript perspective customises the package explorer and the outline view of the current Jadescript code, and it provides the Jadescript code editor. The syntax highlighting support integrated with the Jadescript code editor further enhances the readability of Jadescript codes. Moreover, all the actions that the user can perform on mentioned user interface elements are tailored on the needs of a Jadescript programmer. Finally, new Eclipse wizards are provided to ease the creation of Jadescript projects.

The productivity tools discussed in this paper are intended to fulfil the need for a professional tool to match the expectations of ordinary Java programmers, and of JADE programmers, in particular. Nonetheless, such tools cannot be considered complete and further developments have been already planned. For example, Jadescript semantic classes, defined as part of the Xtext-based compiler, could be extended to implement advanced language tools like an improved validation system with quick fixes and auto-completion actions.

References

[1] J. P. Müller, K. Fischer, Application impact of multi-agent systems and technologies: A survey, in: Agent-Oriented Software Engineering, Springer, 2014, pp. 27–53.

[2] J. B. Larsen, Agent programming languages and logics in agent-based simulation, in: Modern Approaches for Intelligent Information and Database Systems, Springer, 2018, pp. 517–526.

[3] B. Lutati, I. Gontmakher, M. Lando, A. Netzer, A. Meisels, A. Grubshtein, AgentZero: A framework for simulating and evaluating multi-agent algorithms, in: Agent-Oriented Software Engineering, Springer, 2014, pp. 309–327.

[4] S. Monica, F. Bergenti, Location-aware JADE agents in indoor scenarios, in: Proceedings of the 16^{th} Workshop "From Objects to Agents", volume 1382 of *CEUR Workshop Proceedings*, RWTH Aachen, 2015, pp. 103–108.

[5] S. Monica, F. Bergenti, Experimental evaluation of agent-based localization of smart appliances, in: EUMAS 2016, AT 2016: Multi-Agent Systems and Agreement Technologies, volume 10207 of *LNCS*, Springer, 2017, pp. 293–304.

[6] S. Monica, F. Bergenti, Indoor localization of JADE agents without a dedicated infrastructure, in: MATES 2017: Multiagent System Technologies, volume 10413 of *LNCS*, Springer, 2017, pp. 256–271.
[7] S. Monica, F. Bergenti, A comparison of accurate indoor localization of static targets via WiFi and UWB ranging, in: Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection, 2016, pp. 111–123.
[8] S. Monica, F. Bergenti, An optimization-based algorithm for indoor localization of JADE agents, in: Proceedings of the 18^{th} Workshop "From Objects to Agents", volume 1867 of *CEUR Workshop Proceedings*, RWTH Aachen, 2017, pp. 65–70.
[9] S. Monica, F. Bergenti, An experimental evaluation of agent-based indoor localization, in: Proceedings of the Computing Conference 2017, IEEE, 2018, pp. 638–646.
[10] S. Monica, F. Bergenti, Optimization based robust localization of JADE agents in indoor environments, in: Proceedings of the 3^{rd} Italian Workshop on Artificial Intelligence for Ambient Assisted Living (AI*AAL.IT 2017), volume 2061 of *CEUR Workshop Proceedings*, RWTH Aachen, 2017, pp. 58–73.
[11] F. Bergenti, G. Caire, D. Gotta, An overview of the AMUSE social gaming platform, in: Proceedings of the Workshop "From Objects to Agents" (WOA 2013), volume 1099 of *CEUR Workshop Proceedings*, RWTH Aachen, 2013.
[12] F. Bergenti, G. Caire, D. Gotta, Agent-based social gaming with AMUSE, in: Proceedings of the 5^{th} International Conference on Ambient Systems, Networks and Technologies (ANT 2014) and 4^{th} International Conference on Sustainable Energy Information Technology (SEIT 2014), Procedia Computer Science, Elsevier, 2014, pp. 914–919.
[13] F. Bergenti, S. Monica, Location-aware social gaming with AMUSE, in: Y. Demazeau, T. Ito, J. Bajo, M. J. Escalona (Eds.), Advances in Practical Applications of Scalable Multi-agent Systems. The PAAMS Collection: 14^{th} International Conference, PAAMS 2016, Springer International Publishing, 2016, pp. 36–47.
[14] F. Bergenti, G. Caire, D. Gotta, Large-scale network and service management with WANTS, in: Industrial Agents: Emerging Applications of Software Agents in Industry, Elsevier, 2015, pp. 231–246.
[15] F. Bergenti, M.-P. Gleizes, F. Zambonelli (Eds.), Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook, Springer, 2004.
[16] K. Kravari, N. Bassiliades, A survey of agent platforms, Journal of Artificial Societies and Social Simulation 18 (2015) 11.
[17] Y. Shoham, Agent-oriented programming, Artificial Intelligence 60 (1993) 51–92.
[18] Y. Shoham, An overview of agent-oriented programming, in: J. Bradshaw (Ed.), Software Agents, volume 4, MIT Press, 1997, pp. 271–290.
[19] M. Dastani, A survey of multi-agent programming languages and frameworks, in: Agent-Oriented Software Engineering, Springer, 2014, pp. 213–233.
[20] F. Bergenti, G. Petrosino, Overview of a scripting language for JADE-based multi-agent systems, in: Proceedings of the 19^{th} Workshop "From Objects to Agents" (WOA 2018), volume 2215 of *CEUR Workshop Proceedings*, RWTH Aachen, 2018, pp. 57–62.
[21] G. Petrosino, F. Bergenti, An introduction to the major features of a scripting language for JADE agents, in: Proceedings of the 17^{th} Conference of the Italian Association for Artificial Intelligence (AI*IA 2018), volume 11298 of *LNAI*, Springer, 2018, pp. 3–14.

[22] F. Bergenti, S. Monica, G. Petrosino, A scripting language for practical agent-oriented programming, in: Proceedings of the 8^{th} 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), ACM Press, 2018, pp. 62–71.

[23] F. Bellifemine, F. Bergenti, G. Caire, A. Poggi, JADE – A Java Agent DEvelopment Framework, in: Multi-Agent Programming, volume 25 of *MASA*, Springer-Verlag, 2005, pp. 125–147.

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

[25] F. Bergenti, An introduction to the JADEL programming language, in: Proceedings of the 26^{th} IEEE International Conference on Tools with Artificial Intelligence (ICTAI), IEEE Press, 2014, pp. 974–978.

[26] F. Bergenti, E. Iotti, S. Monica, A. Poggi, A case study of the JADEL programming language, in: Proceedings of the 17^{th} Workshop "From Objects to Agents" (WOA 2016), volume 1664 of *CEUR Workshop Proceedings*, 2016, pp. 85–90.

[27] F. Bergenti, E. Iotti, A. Poggi, Core features of an agent-oriented domain-specific language for JADE agents, in: Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection, Springer International Publishing, 2016, pp. 213–224.

[28] F. Bergenti, E. Iotti, S. Monica, A. Poggi, Interaction protocols in the JADEL programming language, in: Proceedings of the 6^{th} ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016) at ACM SIGPLAN Conference Systems, Programming, Languages and Applications: Software for Humanity (SPLASH 2016), ACM Press, 2016, pp. 11–20.

[29] F. Bergenti, E. Iotti, S. Monica, A. Poggi, Overview of a formal semantics for the JADEL programming language, in: Proceedings of the 18^{th} Workshop "From Objects to Agents", volume 1867 of *CEUR Workshop Proceedings*, RWTH Aachen, 2017, pp. 55–60.

[30] F. Bergenti, E. Iotti, S. Monica, A. Poggi, A comparison between asynchronous backtracking pseudocode and its JADEL implementation, in: Proceedings of the 9^{th} International Conference on Agents and Artificial Intelligence (ICAART 2017), volume 2, SciTePress, 2017, pp. 250–258.

[31] F. Bergenti, E. Iotti, S. Monica, A. Poggi, Agent-oriented model-driven development for JADE with the JADEL programming language, Computer Languages, Systems & Structures 50 (2017) 142–158.

[32] R. H. Bordini, J. F. Hübner, M. Wooldridge, Programming multi-agent systems in AgentSpeak using Jason, volume 8, John Wiley & Sons, 2007.

[33] A. S. Rao, AgentSpeak(L): BDI agents speak out in a logical computable language, in: European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Springer, 1996, pp. 42–55.

[34] K. V. Hindriks, F. S. De Boer, W. Van der Hoek, J.-J. C. Meyer, Agent programming in 3APL, Autonomous Agents and Multi-Agent Systems 2 (1999) 357–401.

[35] K. V. Hindriks, F. S. De Boer, W. Van Der Hoek, J.-J. C. Meyer, Agent programming with declarative goals, in: International Workshop on Agent Theories, Architectures, and Languages, Springer, 2000, pp. 228–243.

[36] S. Rodriguez, N. Gaud, S. Galland, SARL: A general-purpose agent-oriented programming language, in: 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), volume 3, IEEE, 2014, pp. 103–110.
[37] R. C. Cardoso, A. Ferrando, A review of agent-based programming for multi-agent systems, Computers 10 (2021) 16.
[38] Eclipse IDE, https://www.eclipse.org/ide, 2021. Accessed on July 8th, 2021.
[39] M. Eysholdt, H. Behrens, Xtext - Implement your language faster than the quick and dirty way tutorial summary, in: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH '10, 2010.
[40] IntelliJ IDEA, https://www.jetbrains.com/idea, 2021. Accessed on July 8th, 2021.
[41] T. J. Parr, R. W. Quong, ANTLR: A predicated-LL(k) parser generator, Software: Practice and Experience 25 (1995).
[42] T. J. Parr, K. Fisher, LL(*): The foundation of the ANTLR parser generator, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2011.