Prototypes of Productivity Tools for the Jadescript Programming Language

Giuseppe Petrosino[a], Eleonora Iotti[a], Stefania Monica[b] and Federico Bergenti[a]

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

Abstract

Jadescript is an agent-based programming language built on top of JADE. Until now, the focus of the development of the language was on design choices, syntax refinements, and the introduction of expressions and constructs for agent-related purposes. In this paper, the crucial goal of making Jadescript suitable for a professional use, as JADE programmers are used to, is first attacked. The success of the Jadescript proposal, as a novel language to build and deploy real-world agent-based systems, is, in fact, necessarily consequent to its usability and integration with mainstream development tools. In particular, the integration of the language in a development environment perfectly fits the daily coding routine of a JADE user, which is an essential aspect toward the successful adoption of the language.

Keywords

Agent Oriented Programming Languages, Agent Oriented Software Engineering, 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 applications, as illustrated, e.g., in the survey [1], where the great impact of agent technologies is discussed. Example applications are agent-based simulations [2], distributed constraints reasoning [3], indoor localization systems [4, 5], just to cite some. In general, multi-agent systems are key instruments for those AI problems when goals to be achieved are sufficiently complex to require the coordination and cooperation of a high number of different agents, often distributed on many hosts, that must use their skills in an effective way in order to gain a collective outcome. 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, agents' norms and validations, and so on. These interesting issues are taken over by *Agent Oriented Software Engineering* (*AOSE*) researcher and in general by agent systems experts, producing a wide range of solutions and frameworks, as well as end-user software platforms [6]. Most, if not all, AOSE methodologies are based on the *Agent Oriented*

WOA 2021: 22^{nd} Workshop "From Objects to Agents", September 2-3, 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)

Programming (*AOP*) [7, 8] paradigm, that explicitly raise the concept of agent as a base building block, ready to be used by the programmer. Many AOP frameworks and platforms, at least the most popular, provide software libraries written in some *General Purpose Languages* (*GPLs*), and such libraries extend the usage of the chosen language to AOP problems. As a matter of fact, a standard way to enrich a GPL with custom functionalities, e.g. agent-based ones, is to provide *Application Programming Interfaces* (*APIs*). Alternatives to the utilization of GPL in software development are various, and its worth citing at least two possibilities, namely, *Domain Specific Languages* (*DSLs*) and scripting languages.

These two alternatives, revisited in the AOSE field, open to a step forward in the direction of AOP, which is the use of *Agent Programming Languages* (*APLs*) [9]. Such languages not only support agent-based features, but put them on a language level. Jadescript, a language built on top of *JADE* (*Java Agent DEvelopment framework*), is an example of AOP scripting language. Other popular APLs are *Jason* [10], which is an implementation of *AgentSpeak(L)* [11], *3APL* [12], *GOAL* [13], *SARL* [14], and so on [15]. The major gain in adopting a novel language for agent development is the presence of native abstractions, constructs, and expressions in such a language that explicitly recall the agent domain, putting agents as first-class entities.

On the other hand, despite the great results in terms of effectiveness and usability, most of the aforementioned APLs has a niche audience, 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, e.g., expert systems, traffic simulation systems, telecommunication routing or networking systems, and so on. A reason for that, among many others, has to be investigated in developers' preferences and programming trends, that, for example, have seen the recent advance of actor-based programming paradigm by means of worldwide popular frameworks such as Akka Actors. 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, though, to bring success to languages that aims at supporting the development of complex environments such as multi-agent systems. The core functionalities offered by an AOP language are appreciated when they are stable and reliable, making it usable for robust applications. This means that the implementation of a novel APL should not only regards the accurate design and implementation of desired capabilities by means of syntactical categories and their semantics, but also the construction of an adequate ecosystem where the language operates. Such an ecosystem could be defined as the set of all tools, utilities, and interfaces that help developers in their daily coding routine, i.e., those services offered by *Integrated Development Environments* (*IDEs*) and middlewares.

In this paper, the main steps taken to build such tools, utilities and interfaces are detailed for the Jadescript language. The discussed approach aims at making the language completely integrated with the Eclipse IDE and its plugins, thus providing tools such as a Jadescript perspective, some specific Eclipse wizards, syntax highlighting, and a launch system for agents. This work had been achievable thanks to another Eclipse plugin, Xtext, which generates some ready and easy-to-use default tools for DSLs. These default tools were then adapted to the agent domain and made it suitable for the Jadescript user experience. The resulting ecosystem is an important step for the growth of the Jadescript language, and brings to it the professional feeling that JADE users expect.

The paper is structured as follows. First, in Section 2 an introduction of the Xtext plugin and related technologies is provided, to give the reader with crucial background information on how to build a professional Eclipse tool. 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 result of this approach is provided to conclude the paper.

2. Xtext Overview

Eclipse Xtext [16] is the main software used to create the discussed tools associated to Jadescript. Xtext presents itself as an open-source framework for development of DSLs and programming languages. It is designed to lift most of the burden of the PL designer, not only by carrying the usual tasks of a parser generator, but also by providing a set of advanced tools that guide in creating a complete compiler and a full-featured IDE.

A Xtext language project is constituted of many Eclipse Java subprojects. Three of them are the most important: (*i.*) the main project, containing the grammar, the semantics and all the other components of the language that are independent of the User Interface (UI); (*ii.*) the `ide` project, which contains the general details and behaviour of the UI (regardless of the specific target environment, so the code in this project can be specialized into, for example, an IntelliJ IDEA [17] plugin, or an editor embedded on a Web page); (*iii.*) the `ui` project, which depends on the `ide` project and contains the specific details related to the Eclipse UI - in other words, it contains the code to implement a custom language plugin for the Eclipse IDE [18].

In the main project, there is the entry point for the language design process, which is the Xtext grammar file of the language. This is because Xtext is *grammar driven*, i.e., there is a grammar language used to generate, from a single source file, all the essential elements of the skeleton of the compiler and the other tools. The first essential element is the lexer, generated from the terminals defined the grammar. The generated lexer is usually complete and sufficient in most of the cases. However, in this case it is worth mentioning, since its default behaviour has been specialized for Jadescript, because it is a language with semantically-relevant indentation. Section 3.2 describes the details of this specialization.

The second element is the generated parser. Predictably, it is used by the compiler and the editor to produce an Abstract Syntax Tree (AST) from the text of the language's source file. With it, a syntax validator is generated by Xtext. This component is in charge of isolating those portions of the code that are syntactically incorrect and of producing the corresponding error messages for the user. Since the Xtext grammar language is an extension of the ANTLR [19] parser generator, the generated parser employs a $LL(*)$ [20] parsing strategy.

Finally, Xtext produces a metamodel of the generated language using *Eclipse Modeling Framework* (EMF) and its metamodel format, *Ecore*. This means that a set of Java interfaces and classes (more specifically, *Java beans*), which constitute a Java object model of the grammar, is created. For each non-terminal grammar rule, a Java *EObject* class is generated, where each component of the rule is mapped to one of the class' properties (i.e., private fields with public accessors). 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 a Xtext grammar and an ANTLR grammar is the ability to add some additional metadata in Xtext to customize the aspects of the generation of the *object model* of the language. Another notable difference is the ability to inject, directly in the model of the generated AST, some useful metadata, like the type of syntax element that a reference can be linked to. This 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 software elements automatically layed out by the Xtext generator from the grammar file, the language designer has already a working editor and other tools working for code editing and syntactical validation. However, to actually interpret and generate executable code, it is required to add some custom components. This is achieved in Xtext by a strong adoption of the Dependency Injection (DI) pattern. The main idea 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 Google Juice's `@Inject`. The Xtext framework uses a *module* class to define the bindings between the Java interfaces of these dependencies with their respective implementations. These bindings are then used by an *Injector* object which creates all the components and their dependencies at runtime. The module class is open for extension and all the binding methods can be overridden, providing the language designer with the ability to change in a controlled and structured way almost any aspect of their language and its associated tools.

Among these aspects, two require particular attention, namely code validation and compiled code generation. Their implementations are the main components of the software that mostly depend on the semantics of the language.

Semantic validation of the user code is managed by a *validator* class. In Xtext, such a class is implemented with a declarative approach - the language designer has to specify, by adding methods to the class, which types of the Java object model of the AST he wishes to check. In case of erroneous or problematic code, these methods can build and emit errors, warnings and informational markers, bound to specific regions of the source code, that are used as feedback with text messages.

Compiled code generation can be achieved by means of a class that extends the interface *IGenerator*, which defines how to generate new files starting from the AST obtained from parsing a source file. For JVM-based languages, however, another specialized approach is available, based on the *JVMModelInferrer* facility. By extending this class, the user is able to symbolically declare which Java classes, interfaces, methods and fields are generated from each source file. Xtext keeps track of these mappings and exploits them to ease the implementation of the language by partially generating: (*i.*) a type checking system, based on the Java type system; (*ii.*) a scope provider, that is able to resolve references to Java packages, types and symbols; (*iii.*) a code generator that creates the `.java` files corresponding to the declared Java structures provided by the JVMModelInferrer; (*iv.*) some IDE features like basic autocompletion, symbolical navigation in the editor, linking between written and generated code and basic refactoring. Of course, if the validator or the type checker find errors in the source code, the compiler aborts code generation, signaling the problem to the user.

Jadescript is currently a language based on the JVM. More specifically, it is compiled to Java code. For this reason, the Jadescript compiler uses JVMModelInferrer facility to take advantage of the pre-built 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 schematized in Fig. 1.

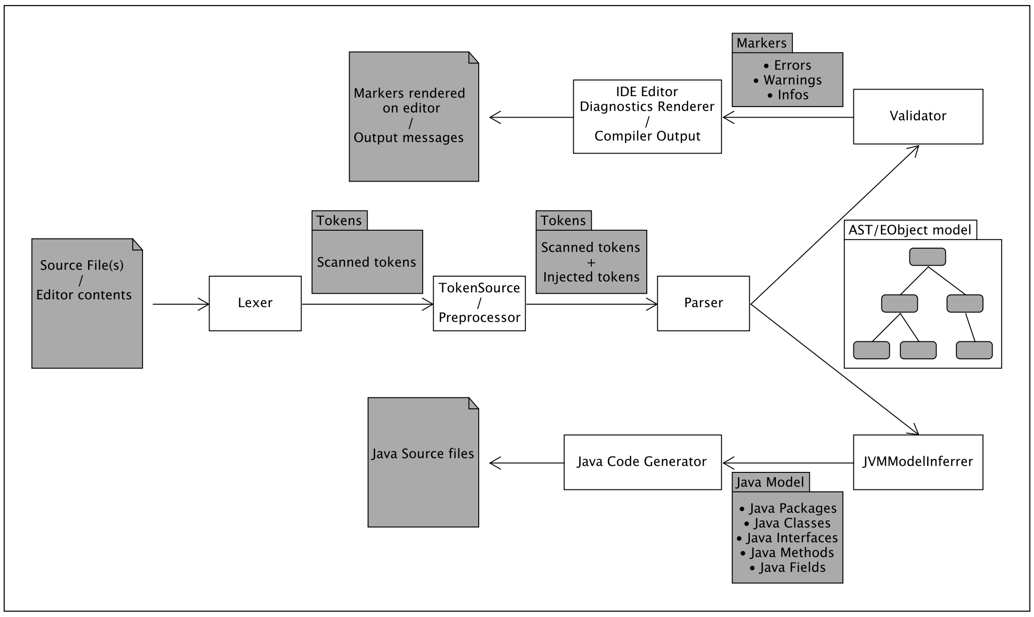


Figure 1: The compilation and validation processes in the Xtext framework for JVM-based languages.

When the user feeds the compiler with a set of Jadescript source files, or when they save some changes in the editor, 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 facility, provided by Xtext, is used to preprocess this stream of tokens, in order to inject into the stream some synthesized tokens relative to the semantically relevant indentation. The new stream of tokens is fed to the parser, which produces an AST and a EMF model of it. These results are then fed to the validator, which statically analyzes the code in search for problems, following the semantics 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, 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 compiled code.

3. 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 agent platforms. The plugin was created with the help of Xtext, especially for those features of it strongly connected to the syntax and semantics of the language, and that, therefore, compose the Jadescript compiler. However, some of the features were created by means of the tools provided by the Eclipse Plugin Development Environment (PDE).

3.1. Xtext Extensions

As mentioned in section 2, in Xtext, new language projects enjoy of a set of great features implemented by default and generated automatically from the grammar file. However, some aspects require the language developer attention to adapt the default Xtext implementations by overriding them with custom code. For Jadescript two aspects required specialized implementations, namely the *TokenSource* class 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 a Xtext-generated parser, the parser reads the input tokens from a *TokenSource*, which is an object that produces tokens from source code when required by the parser. In most programming languages, the parser includes the assumption that whitespace tokens (i.e.: blanks, tabulation, new line characters) are hidden in the grammar and not considered in the input stream of tokens, since their main purpose is to act as separators between parts of the text that are relevant according the grammar. This is not completely true for languages with semantically relevant indentation[1] like Jadescript. In those 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 the kind of instruction expressed by the line and how that 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 these 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 new line character as hint. In this inference mechanism, the compiler has to leave the possibility for the user to split in an orderly fashion any statement in two or more lines, whenever such statements are too long and the programmer whishes to increase readability.

In Jadescript, this set of behaviours is encoded into a special kind of tokens (also known as *synthetic tokens*) that signals the parser for the occurrence three types of structural points in the code: (*i.*) the point of termination of a statement (*NEWLINE* synthetic token); (*ii.*) the point where a new code block is opened (*INDENT* synthetic token); (*iii.*) 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 which 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 one (or several) new line characters is encountered in the stream, the algorithm 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`,

[1]and for some other esoteric exceptions, like *Whitespace* [21]

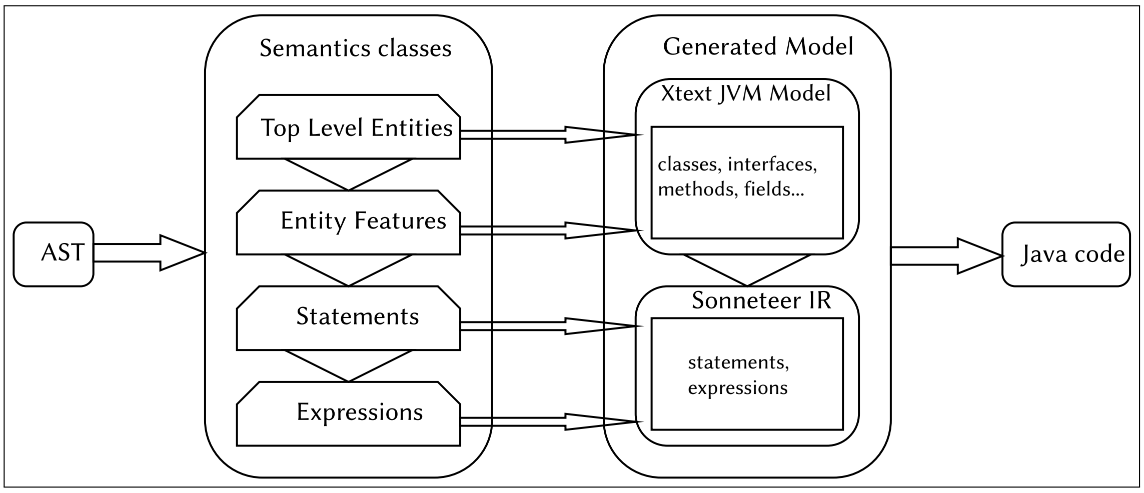


Figure 2: The semantics classes categories and how their generated artifacts are composed.

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 algorithm 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, signaling 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. This set of rules is designed to take advantage of the Jadescript programmer's intuition in using the indentation and of the line breaks in its code to intelligently infer some structural aspects of the agent program.

3.1.2. Jadescript Semantics 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 *semantics classes*. Each one of these classes handles how a particular node of the AST is used for code generation and validation.

The semantics 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 semantics 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` facility generated by Xtext.

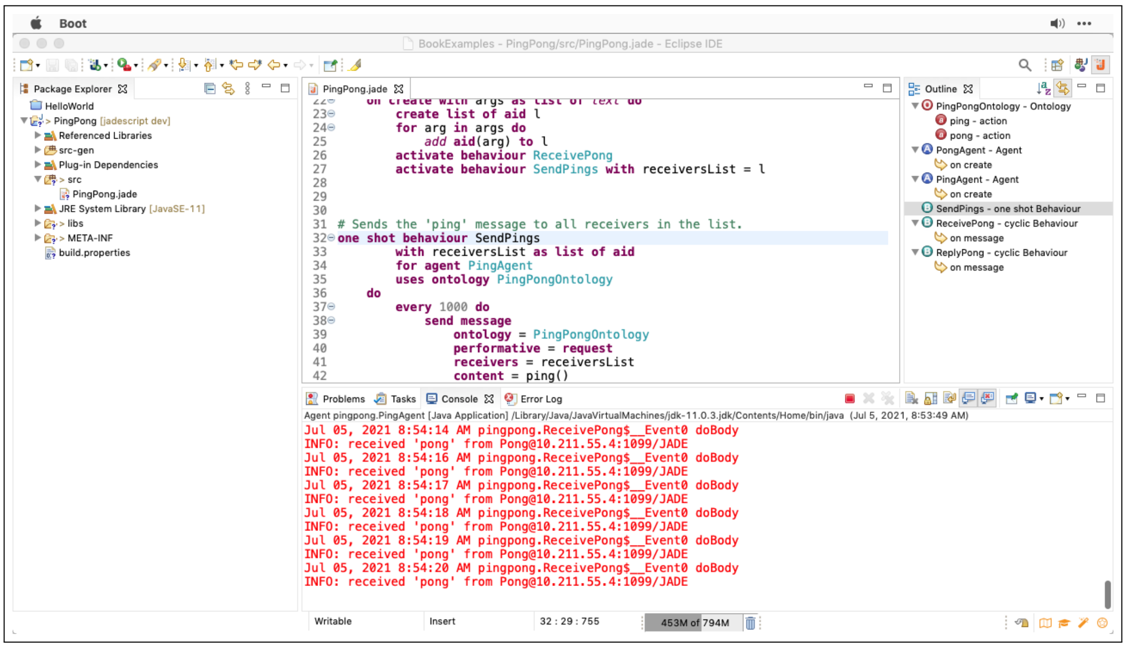
Entity Features. These elements of the language are the main building blocks of each top-level declaration. They are usually directly enumerated in the declaration's body, and for this reason, they appear as indented by just one level in the source code. Examples of features include *event handlers* 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.

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 semantics classes into objects of a custom Internal Representation (IR) model, named *Sonneteer*. Sonneteer is a small Java library developed by AILab which implements a simple API to generate text strings of Java source code. The usage of this library in the implementation of the semantics 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. As a matter of fact, each semantics class related to type of expression includes a set of methods to compute the type of the expression. Semantics classes related to this category compile expressions into simple text strings, but they do it in three steps. In the first step, all the statements and expressions in a procedural code block are translated into Sonneteer objects, which include some placeholder elements which annotate the generated code with some compiler metadata. These placeholder elements are then analyzed in the second step. The result of the analysis is finally used in the third step to perform some optimizations and late-linking between variables. Note that expressions include literals, infix and unary operations like addition and logical negation, and other particular operations like `matches` for pattern matching.

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 customization of functionality. Such customizations can be created in plug-ins, and the extensibility approach



```
    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 an Eclipse IDE workspace with the Jadescript perspective.

allows plug-ins to use and extend other plug-ins declaring a set of hooks in the *extension points* in the plugin's manifest XML file. The Jadescript plugin uses these extension points to customize 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 center, 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 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.

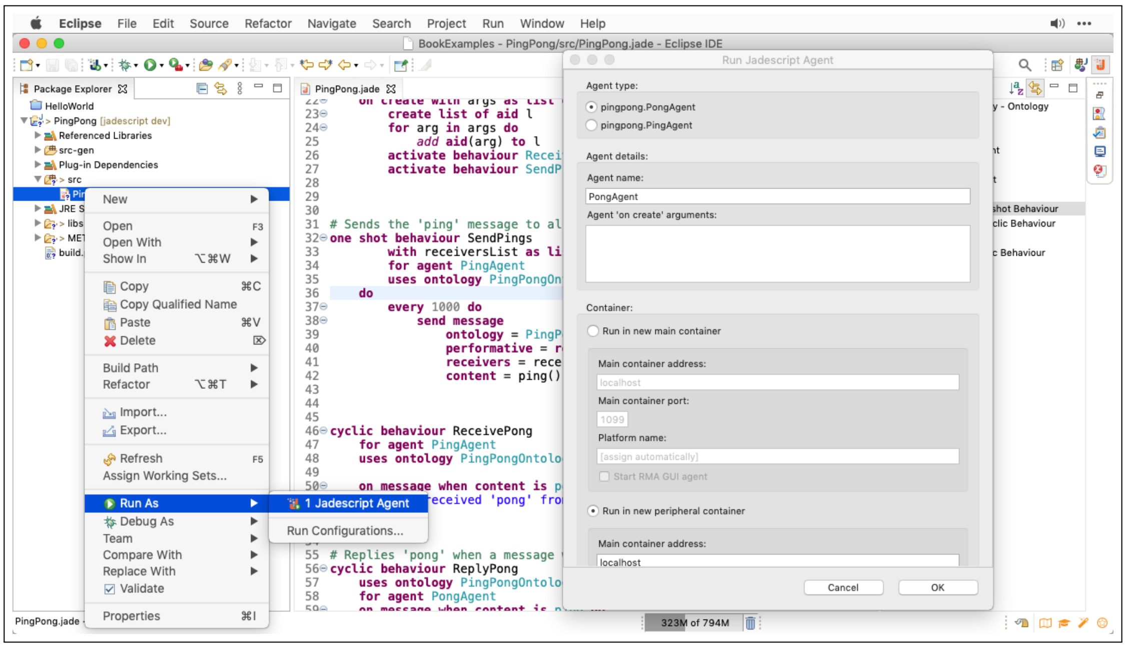
Four more wizards are used to guide the user to create new Jadescript source files. The *New Jadescript File* wizard creates a new empty Jadescript file in the specified project location, with the specified module. This wizard is the specialized into the *New Jadescript Agent*, *New Jadescript Ontology* 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 colors, in order to help the user quickly recognize and tell the various elements of the code. This aspect of the appearance of code in the Jadescript editor can be customized 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 semantics rules to highlight the text of different colors, and this is done by using a set of special methods in the semantics classes. This approach allows, for example, to highlight with different colors 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 plugin's extension points implement two actions in the IDE. The first is accessible from the Eclipse's 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 button is a pulldown button, with a second option available in its dropdown menu. By clicking on the second option, the created JADE main container includes a *Remote Monitoring Agent* (RMA) with a Graphical User Interface (GUI) 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



```
        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
                 received 'pong' fro

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

Run Jadescript Agent

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 Eclipse.

currently open project's Java classpath. In this way, from within the RMA GUI it is possible to spawn 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 itemizes a list of selectable agent types, which correspond to the ones declared in the source file. The dialog then allows the user to customize 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, programming tools and user interfaces explicitly designed for Jadescript users are discussed. The approach is driven by the idea of moving toward a professional ecosystem for the novel AOP language Jadescript, taking into account developer preferences and trends as well as the advantages provided by modern IDE facilities. As a base building block of the internal development of Jadescript, Xtext was employed. The brief Xtext overview presented in this paper detailed its functioning and explained the importance of such a framework not only in the generation of Java sources but also clarified its role in the validation of code and generation of errors, warnings, and information, useful to the programmer who aims to use Jadescript. The Eclipse IDE Plugin for Jadescript is then discussed, with a slight digression on the token source system as a Xtext extension. Such an extension is of fundamental value since it introduces semantically relevant indentation, and the implementation of such a kind of indentation in Jadescript also allows users to split long lines. These could be seen as small details, but they play an important role to greatly improve readability and to give a modern look-and-feel to the language. Apart from that, the users can also take advantage of all the utilities that the Eclipse IDE has for their other project, such as Java ones. In particular, the Jadescript perspective customizes the package explorer, the outline with a structure view of the semantic entities of Jadescript used in the code, and the main space, which is taken by the actual editor. Actions on these spaces are tailored on the needs of a multi-agent system developer, providing in particular a way to launch containers and Jadescript agents, thus easing the integration with JADE. Moreover, Eclipse wizards were introduced to ease and speed up the creation of the Jadescript project and the syntax highlighting further enhance the usability of the language. All these environment-related features together fulfill the need for a complete and professional tool that matches the expectations of JADE users. As future works, Jadescript semantic classes, defined as part of the Xtext compiler, can also be extended and used to provide more advanced language tools, like an improved validation system with *quickfix* actions for the user and an autocompletion system working in the Eclipse IDE editor for Jadescript.

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] 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.

[5] 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.

[6] K. Kravari, N. Bassiliades, A survey of agent platforms, Journal of Artificial Societies and Social Simulation 18 (2015) 11.

[7] Y. Shoham, Agent-oriented programming, Artificial Intelligence 60 (1993) 51–92.

[8] Y. Shoham, An overview of agent-oriented programming, in: J. Bradshaw (Ed.), Software Agents, volume 4, MIT Press, 1997, pp. 271–290.

[9] M. Dastani, A survey of multi-agent programming languages and frameworks, in: Agent-Oriented Software Engineering, Springer, 2014, pp. 213–233.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] R. C. Cardoso, A. Ferrando, A review of agent-based programming for multi-agent systems, Computers 10 (2021) 16.

[16] 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. doi:10.1145/1869542.1869625.

[17] IntelliJ IDEA, https://www.jetbrains.com/idea/, 2021. Accessed on July 8th, 2021.

[18] Eclipse IDE, https://www.eclipse.org/ide/, 2021. Accessed on July 8th, 2021.

[19] T. J. Parr, R. W. Quong, ANTLR: A predicated-LL(k) parser generator, Software: Practice and Experience 25 (1995). doi:10.1002/spe.4380250705.

[20] T. 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. doi:10.1145/1993498.1993548.

[21] E. Brady, C. Morris, Whitespace, https://web.archive.org/web/20150523181043/http://compsoc.dur.ac.uk/whitespace/index.php, 2003. Accessed on May 23rd, 2015.