Overview of a Formal Semantics for the JADEL Programming Language

Federico Bergenti*, Eleonora Iotti†, Stefania Monica* and Agostino Poggi†
* Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Università degli Studi di Parma
Parco Area delle Scienze 53/A, 43124 Parma, Italy
Email: federico.bergenti@unipr.it, stefania.monica@unipr.it
† Dipartimento di Ingegneria e Architettura
Università degli Studi di Parma
Parco Area delle Scienze 181/A, 43124 Parma, Italy
Email: eleonora.iotti@studenti.unipr.it, agostino.poggi@unipr.it

***Abstract*—This paper outlines a first proposal of a formal semantics for the JADEL programming language. JADEL is an agent-oriented programming language based on JADE that has been recently proposed to ease the adoption of JADE, and to promote its use. In previous works, JADEL was specified at the syntax level, and only an informal semantics was given. The major contribution of this paper is to outline a formalization of the semantics of JADEL to complement previous works and to allow reasoning on JADEL agents and multi-agent systems. First, the paper provides a brief recall on JADEL by describing its main abstractions and their specific syntactic constructs. Then, a discussion of the proposed operational semantics based on transition systems is described. Finally, a detailed operational semantics of only some relevant constructs is given. The validity of the proposed approach is discussed in the conclusion of the paper, together with directions of future developments.**

I. Introduction

JADEL, which stands for *JADE* (*Java Agent DEvelopment framework*) *Language*, is an agent-oriented programming language designed to help the development of JADE [1] agents and multi-agent systems. A preliminary work on JADEL [2] shows the main ideas and motivations behind its creation, together with a first example of JADEL code. More recent works [3] describes the current state of the development of JADEL and related tools, and frame JADEL in the scope of model-driven development. In short, JADEL was conceived to meet the needs of software developers that want to take advantage of agent technologies—and of JADE in particular—with no need to deal with the implementation details that ordinary use of JADE requires. Actually, writing complex JADE applications is sometimes perceived as a difficult task, especially by developers who are starting to approach it. Due to its inherent complexity and its continuous growth, JADE has now a steep learning curve, especially for the number of low level implementation details that the developer is required to master. Despite these difficulties, JADE is widely recognized as one of the most popular tools to develop multi-agent systems, and it is used successfully in very different contexts, from academic research to industrial applications [4], in the constant attempt to effectively use the beneficial features of agents from the point of view of software development [5]. Moreover, related projects *WADE* (*Workflows and Agents Development Environment*) [6]–[8], and *AMUSE* (*Agent-based Multi-User Social Environment*) [9], [10] contributed to increment the possibilities of JADE. JADE is now a complete tool, but such a completeness comes at a cost: its constantly increasing complexity. One of the reasons for the initial success of JADE is that it was designed as a Java library, which was an appreciated choice in the early 2000s. At the time, Java was quickly becoming one of the most promising technologies, and developers wanted to use it, also because it was tightly connected with the growth of the Web. In fact, Java, and the long gone Java applets, played a key role in the rapid expansion of the Web. The idea of supporting developers of multi-agent systems with a Java library, rather than with a specific language, is one of the fundamental design choices behind JADE. But, nowadays, a pure Java approach is less appealing because a number of valid alternatives are becoming popular, and, at the same time, because *DSLs* (*Domain-Specific Languages*) are becoming popular among developers, and not just among domain experts [11], [12].

The work on JADEL started [2] to provide current and future JADE users with a simpler, yet seamlessly effective, tool. The major purpose of JADEL is to provide a high-level view on the most important abstractions that JADE provides, allowing developers to concentrate on agent-oriented abstractions, rather than on lower-level details. JADEL is by design a DSL, whose host language is a dialect of Java called *Xtend* [13]. The choice of developing a DSL is not accidental, since DSLs are often simpler to learn and to use, thanks to their lighter syntax, and because they are tailored on the specific needs of their target domain, which is agent-oriented programming in this case. Well-designed DSLs provide a small number of relevant abstraction, constructs and expressions, whose purpose is to manage effectively the specific abstractions of their specific domains.

The core features of JADEL, namely agents, ontologies, and behaviours, are briefly described in [14], and the recent extension of the language to support roles in FIPA interaction

protocols (see, e.g. [15]) is discussed in [16], [17]. A nontrivial example of a JADEL multi-agent system is presented in [3]. Such an example is the implementation of the *asynchronous backtracking* [18] algorithm, which is used as a test case to evaluate the expressiveness of JADEL against an informal pseudocode [3], and which is also used to validate the possibility of reinterpreting relevant applications of other parallel and distributed computing paradigms in terms of agents (see, e.g., [19], [20]). In all those previous works, the semantics of JADEL was described only informally, and this paper complements those works by describing the principal parts of an operational semantics for JADEL. Obviously, there are relevant advantages in specifying a formal semantics for a programming language, for example, to support verification and compiler implementation. Unfortunately, general-purpose languages, such as Java, are too difficult to formalize completely, and only minimal extracts are formalized (see, e.g., [21]). On the contrary, DSLs are typically sufficiently small, and for some of them a complete formalization was provided. Moreover, agent-oriented programming languages are often provided with a formal semantics, and for some of them the formalization predated the implementation of tools, e.g., AgentSpeak(L) [22], and Concurrent MetateM [23]. For other agent-oriented programming languages, the formal semantics came together with the implementation of tools, e.g., *SEA_L* (*Semantic web-Enabled Agent Language* [24], and *SEA_ML Semantic web-Enabled Agent Modeling Language* [25]. Notably, relevant studies intended to provide JADE with a formal semantics are available, e.g. [26], and an overview of a complete formalization of JADE in terms of transition systems can be found in [27], [28].

This paper is organized as follows. Section II briefly describes the main abstractions and constructs of JADEL. Section III provides a summary of the syntax of the language. Section IV shows the most relevant parts of an operational semantics for JADEL. Finally, Section V concludes the paper and discusses future developments.

II. Overview of JADEL

JADEL is an agent-oriented programming language designed around the features of *Xtext* [29], a framework which provides effective support for the development of DSLs. The use of Xtext eases the design of a DSL because it simplifies the main steps involved in such a task, e.g., the creation of the grammar and the implementation of the compiler. First, Xtext provides a DSL to express *EBNF* (*Extended Backus-Naur Form*) grammars, from which a parser can be easily obtained with the help of a parser generator. Then, Xtext provides a base grammar, called *Xbase* grammar [30], which is highly extensible and it is used to implement the basic features of the *Xtend* language [13], such as expressions, and type references. Xtend is a dialect of Java, and its syntax and semantics rely on those of Java, but specific syntactic facilities are provided to make it lighter and simpler. JADEL can be considered an agent-oriented extension of Xtend.

JADE provides a number of abstractions, and related Java classes, for the construction of agents and multi-agent systems. JADEL selects only a few primary abstractions among them in order to provide the developer with an agent-oriented view of agents and multi-agent systems. Only four main abstractions that JADE implements were chosen, namely agents, behaviours, communication ontologies, and interaction protocols. For the sake of brevity, and because the support for interaction protocols is still at an early stage, in this paper only agents, behaviours and ontologies are considered. In detail, JADEL agents use ontologies and behaviours, and the syntax of JADEL clearly highlights the connections of the agent with ontologies and behaviours. The declaration of an agent is allowed to extend the declaration of another agent, with the usual semantics of inheritance, and two event handlers are provided to support initialization and take-down phases. Behaviours can be activated in such initialization and take-down phases by means of specific expressions. Actually, JADEL provides a specific syntax to declare and activate behaviours, and it also offers specific constructs to manage actions and events.

The behaviours of JADEL can be `cyclic` or `oneshot`, and the semantics of such types of behaviour is the same as JADE cyclic and one-shot behaviours, respectively. A behaviour can be specific to a group of agents, i.e., it can take advantage of the common characteristics of such agents in the definition of its action. Also, a behaviour can refer to a specific communication ontology. The body of a behaviour contains a set of fields, a set of methods and a nonempty set of event handlers. In fact, at least one event handler must be present in order to define the action of the behaviour. Event handlers are specified for behaviours by means of the construct `on-when-do` which identifies the event, states conditions on it, and describes the action to perform in response. Behaviours can extend other behaviours, with the usual semantics of subclassing, and all event handlers of a base behaviour are added to all derived behaviours.

The ontologies of JADEL are formal means to support the semantics of agent communication languages for specific problems. An ontology provides a dictionary of terms and schemas, which can be arranged in a hierarchy. Terms and schemas are used to send and receive syntactically-correct messages. In detail, an ontology consists of a set of propositions, a set of predicates, and a set of concepts, which can be basic or composite. Propositions are first-order logics well-formed formulas. Predicates are first-order logics predicates with an arity, and their arguments are terms formed using concepts. Basic (or atomic) concepts are atomic terms provided by JADE. They can be composed to create other (composite) concepts, which can be used to express complex terms. Composite concepts can be seen as function symbols in first-order logics. They are terms with arguments, and such arguments are terms themselves. Predicates are used to state relations among concepts, while concepts are used to to describe entities of the domain. Both concepts and predicates can be derived from other base concepts and predicates, respectively.

$$
\begin{array}{lcl}
odecl & ::= & \texttt{ontology}\ o\ \overline{\texttt{extends}\ o_{base}}^{?} \\
 & & \{\ \overline{propdecl}^{*}\ \overline{cdecl}^{*}\ \overline{pdecl}^{*}\ \} \\
propdecl & ::= & \texttt{proposition}\ prop \\
cdecl & ::= & \texttt{concept}\ c\ (\ \overline{cpar}^{*}\ )\ \overline{\texttt{extends}\ c_{base}}^{?} \\
pdecl & ::= & \texttt{predicate}\ p\ (\ \overline{cpar}^{*}\ )\ \overline{\texttt{extends}\ p_{base}}^{?} \\
cpar & ::= & \overline{\texttt{many}}^{?}\ c\ x \mid \overline{\texttt{many}}^{?}\ cbasic\ x \\
cbasic & ::= & \texttt{aid} \mid \texttt{bool} \mid \texttt{byte\_sequence} \\
 & \mid & \texttt{content\_element\_list} \mid \texttt{date} \mid \texttt{float} \\
 & \mid & \texttt{integer} \mid \texttt{string}
\end{array}
$$

Fig. 1. JADEL grammar for ontologies. Metavariables o, o_{base} denote ontologies, $prop$ denotes a proposition, c, c_{base} denote concepts, p, p_{base} denote predicates, and x denotes a generic variable.

$$
\begin{array}{lcl}
expr & ::= & xexpr \mid actb \mid extr \mid send \\
actb & ::= & \texttt{activate behaviour}\ \overline{x\ \texttt{as}}^{?}\ b(\overline{xexpr}^{*}) \\
extr & ::= & \texttt{extract}\ x\ \texttt{as}\ t \\
send & ::= & \texttt{send message}\ \overline{m}^{?}\ \{\ \overline{msgexpr}^{*}\ \} \\
msgexpr & ::= & pexpr \mid oexpr \mid cexpr \mid rexpr \\
pexpr & ::= & \texttt{performative is INFORM} \mid \ldots \\
oexpr & ::= & \texttt{ontology is}\ o \\
cexpr & ::= & \texttt{content is}\ x \\
rexpr & ::= & \texttt{receivers are}\ l
\end{array}
$$

Fig. 2. JADEL grammar for extended Xtend expressions. $xexpr$ refers to standard Xtend expressions, metavariables x, y denote variables, b denotes a behaviour, t denotes the name of a type, m denotes a message, o denotes an ontology, and l denotes a variable which refers to a list of agent identifiers.

III. The Grammar of JADEL

This section summarizes the grammar of JADEL to support the description of an operational semantics in next section. A detailed description of the grammar of JADEL is included in an upcoming paper. The syntax that JADEL adopts is provided starting from ontologies, whose grammar is shown in Figure 1 using the EBNF language, where

1) $\overline{X}^{*}$ stands for the repetition of X zero or more times;
2) $\overline{X}^{+}$ means that X is repeated one or more times; and
3) $\overline{X}^{?}$ means that X is optional.

Figure 2 shows the grammar of JADEL expressions, as an extension of the grammar of Xtend expressions. In fact, it is worth noting that JADEL relies on Xtend expressions instead of introducing a new syntax for expressions. This choice has the advantage of grounding JADEL on a solid grammar for expressions whose primary goal is to support the construction of procedural languages. The syntax of extended expressions introduced in Figure 2 provides specific features to activate behaviours, to send messages and to extract the content of messages. For JADEL, messages are structures that have a number of fixed properties: the performative, the list of recipients, the ontology and the content. The performative denotes the type of the message, and exactly one performative is contained in a syntactically-correct message. The list of recipients specifies the agent identifiers of all agents that are intended to receive the message. Ontologies are identified by their names, and they must be declared using the grammar of Figure 1. Finally, the content of a message can be either a proposition, a concept, a predicate, a string of characters or a sequence of bytes.

$$
\begin{array}{lcl}
bdecl & ::= & btype\ \texttt{behaviour}\ b\ (\overline{t\ x}^{*})\ \overline{\texttt{for}\ a}^{?}\ \overline{onto}^{?} \\
 & & \overline{\texttt{extends}\ b_{base}}^{?} \\
 & & \{\ \overline{field}^{*}\ \overline{method}^{*}\ \overline{bevent}^{+}\ \} \\
btype & ::= & \texttt{cyclic} \mid \texttt{oneshot} \\
onto & ::= & \texttt{uses ontology}\ o \\
field & ::= & (\texttt{var} \mid \texttt{val})\ \overline{t}^{?}\ f\ \overline{= expr}^{?} \\
method & ::= & t\ m(\overline{t\ mpar}^{*})\ \{\ \overline{expr}^{*}\ \} \\
bevent & ::= & \texttt{do}\ \{\ \overline{expr}^{*}\} \\
 & \mid & \texttt{on message}\ m\ \overline{\texttt{when}\ \{wexpr\}}^{?} \\
 & & \texttt{do}\ \{\ \overline{expr}^{*}\} \\
wexpr & ::= & wexpr\ \texttt{or}\ wexpr \mid wexpr\ \texttt{and}\ wexpr \\
 & \mid & \texttt{not}\ wexpr \mid pexpr \mid oexpr \mid cexpr
\end{array}
$$

Fig. 3. JADEL grammar for behaviours. Metavariables b, b_{base} denote behaviours, t denotes a type, x denotes a variable, a denotes an agent type, o denotes an ontology, f denotes the name of a field, m denotes the name of a method, $mpar$ denotes the name of a parameter, m denotes a message, while $expr$, $cexpr$, $pexpr$, and $cexpr$ are defined in Figure 2.

$$
\begin{array}{lcl}
adecl & ::= & \texttt{agent}\ a\ \overline{onto}^{?}\ \overline{\texttt{extends}\ a_{base}}^{?} \\
 & & \{\ \overline{field}^{*}\ \overline{method}^{*}\ \overline{aevent}^{+}\ \} \\
aevent & ::= & \texttt{on}\ event\ \{\ \overline{expr}^{*}\ \} \\
event & ::= & \texttt{create} \mid \texttt{destroy}
\end{array}
$$

Fig. 4. JADEL grammar for agents. Metavariables a, a_{base} denote agents, while $onto$, $field$, $method$ and $expr$ are defined in Figure 2.

Figure 3 shows the JADEL grammar for behaviours. In JADEL, the reception of a message is an event, and, as said previously, it corresponds to the use of a specific construct `on-when-do`, called $bevent$ in Figure 3. Such a construct captures the event corresponding to an incoming message, and it can express conditions on incoming messages by means of message templates defined in the `when` block. Message templates refer to the properties of incoming messages, and they can be combined using logic connectives. The closing block `do` contains the block of code to be executed when conditions on messages hold.

Finally, JADEL grammar for the declaration of agents is shown in Figure 4. As expected, it simply let the developer manage the lifecycle of agents, which is where behaviours are activated or deactivated.

IV. An Operational Semantics for JADEL

The semantics of JADEL briefly described in this paper is formally defined by means of operational rules and auxiliary lookup functions. The three lookup functions from the operational semantics of *Featherweight Java* (*FJ*) [21], namely $fields$, $mtype$ and $mbody$, are used to connect the semantics of the agent-oriented features of JADEL with the semantics of the host language, which is nothing but a syntactic dialect of Java. Actually, the agent and behaviour abstractions that JADEL provides are mapped into Java classes that derives

$$\frac{\texttt{agent}\ A_1\ \texttt{extends}\ A_2\{\overline{F}^*\ \overline{M}^*\ \overline{aevent}^+\} \quad fields(A_2) = \overline{G}^*}{fields(A_1) = \overline{F}^*\ \overline{G}^*} \tag{1}$$

$$\frac{\texttt{agent}\ A_1\ \texttt{uses ontology}\ O_1\ \texttt{extends}\ A_2\{\overline{F}^*\ \overline{M}^*\ \overline{aevent}^+\} \quad ontologies(A_2) = \overline{O}^*}{ontologies(A_1) = O_1\ \overline{O}^*} \tag{2}$$

$$\frac{\texttt{agent}\ A\ \ldots\ \{\overline{F}^*\ \overline{M}^*\ \overline{aevent}^+\} \quad \texttt{on create}\ \{\overline{expr}^*\} \in \overline{aevent}^+}{mtype(\texttt{setup}, A) = \epsilon \to \epsilon \quad mbody(\texttt{setup}, A) = \langle \epsilon, \overline{expr}^* \rangle} \tag{3}$$

$$\frac{\texttt{agent}\ A\ \ldots\ \{\overline{F}^*\ \overline{M}^*\ \overline{aevent}^+\} \quad \texttt{on destroy}\ \{\overline{expr}^*\} \in \overline{aevent}^+}{mtype(\texttt{takeDown}, A) = \epsilon \to \epsilon \quad mbody(\texttt{takeDown}, A) = \langle \epsilon, \overline{expr}^* \rangle} \tag{4}$$

Fig. 5. Rules that specify the operational semantics of agents.

$$\frac{BT\ \texttt{behaviour}\ B_1(\overline{t\ x}^*)\ \texttt{extends}\ B_2\{\overline{F}^*\ \overline{M}^*\ \overline{bevent}^+\} \quad fields(B_2) = \overline{G}^*}{fields(B_1) = \overline{t\ x}^*\ \overline{F}^*\ \overline{G}^*} \tag{5}$$

$$\frac{BT\ \texttt{behaviour}\ B_1(\overline{t\ x}^*)\ \texttt{for}A\ \texttt{extends}\ B_2\{\overline{F}^*\ \overline{M}^*\ \overline{bevent}^+\} \quad fields(B_2) = \overline{G}^*}{fields(B_1) = \overline{t\ x}^*\ \overline{F}^*\ \overline{G}^*\ a} \tag{6}$$

where a is the field A $\texttt{theAgent} = (A)$ $\texttt{myAgent}$;

$$\frac{BT\ \texttt{behaviour}\ B_1\ \texttt{extends}\ B_2\{\ldots\overline{bevent}^+\} \quad events(B_2) = \overline{bevent_2}^+}{events(B_1) = \overline{bevent}^+\ \overline{bevent_2}^+} \tag{7}$$

Fig. 6. Rules that specify the operational semantics of behaviours.

from classes `Agent` and `Behaviour`, respectively. Such classes are provided by JADE in its API and, obviously, they have fields and methods. In detail, in the operational semantics of FJ, the $fields$ lookup function associates each class name with its own fields plus inherited fields. For JADEL agents, $fields$ works exactly as in FJ, as shown by rule (1) in Figure 5. Despite this, there are differences between FJ classes and JADEL agents and behaviours. For example, the two agent event handlers `on-create` and `on-destroy` implicitly provide two methods, as shown in Figure 5, rules (3) and (4), which are not part of FJ.

Methods are identified by means of the two functions $mtype$ and $mbody$. The first function takes the name of the method and the name of the class, and returns a mapping between the parameter types and the return type of the method. When the return type is `void`, or there are no parameters, we conventionally use ϵ. The second function, $mbody$, also takes the name of the method and the name of the class, and it returns a pair, whose first element is a list of parameters, and whose second element is the actual body of the method. The definition of $mtype$ and $mbody$ for JADEL agents and behaviours is the same that of FJ. Behaviour fields, instead, are obtained not only by the user declared fields, but also by behaviour parameters, and there is an implicitly declared field `theAgent`, which identifies the agent that is currently using a behaviour, as shown in Figure 6, rules (5) and (6).

Two additional auxiliary lookup functions are defined for ontologies and events. Function $ontologies$ takes an agent and it returns a list of ontologies, as in rule (2), when an ontology is specified by the declaration `uses-ontology`. Function $events$, instead, is defined only for behaviours, and it maps the name of a behaviour with its list of declared events. It is worth noting that the list of events is not limited to the event handlers that are specified in the behaviour, but it also contains inherited events, as shown in rule (7).

The management of events also requires the definition of such inherited events, even if they are not JADEL abstractions, at least explicitly. In fact, in JADEL, event handlers are translated into inner classes of the host behaviour, and they are composed of specific fields and methods, which collectively define the actual action of the behaviour.

For the sake of brevity, only some rules to manage events are shown in Figure 7. Rules (8) and (9) define the $innerclasses$ lookup function, which takes the name of a behaviour and a list of events, and it returns a pair whose first element is the definition of the current inner class plus the already defined inner classes, and whose second element is the number of the processed events. Each inner class `Event`n has a list of fields and methods, which are identified by looking at the definition of the event handler. As shown in rule (10), if the event handler is in the form of the `on-when-do` construct, two fields are implicitly defined, namely, an `ACLMessage` field and a `MessageTemplate` field. Three methods of `Event`n are also defined, namely the `receive`, `doBody` and `run`. Those are all `void` methods without parameters. Rule (11) shows the definition of the `receive` method. The `doBody` method

$$\frac{\overline{bevent}^{*} = \emptyset}{innerclasses(B, \overline{bevent}^{*}) = \langle \epsilon, 0 \rangle} \tag{8}$$

$$\frac{bevent_n \in events(B) \quad innerclasses(B, \overline{bevent}^{*}) = \langle E, n \rangle}{innerclasses(B, bevent_n\ \overline{bevent}^{*}) = \langle \texttt{private class Event}n\{\overline{F}^{*}\ \overline{M}^{*}\}\ E, n+1 \rangle} \tag{9}$$

where $\overline{F}^{*}$ and $\overline{M}^{*}$ depend on $bevent_n$

$$\frac{bevent_n = \texttt{on message}\ m\ \texttt{when}\{wexpr\}\texttt{do}\{\overline{expr}^{*}\}}{fields(\texttt{Event}n) = \texttt{ACLMessage}\ m;\ \texttt{MessageTemplate}\ mt = wexpr;} \tag{10}$$

$$\frac{bevent_n = \texttt{on message}\ m\ \texttt{when}\{wexpr\}\texttt{do}\{\overline{expr}^{*}\}}{mbody(\texttt{receive}, \texttt{Event}n) = \langle \epsilon, m = \texttt{theAgent.receive(mt)}; \rangle} \tag{11}$$

$$\frac{events(B) = bevent_0 \ldots bevent_N}{mbody(\texttt{action}, B) = \langle \epsilon, \texttt{super.action()}; \texttt{Event0.run()};\ \ldots\ \texttt{Event}N\texttt{.run()}; \rangle} \tag{12}$$

Fig. 7. Rules that specify the operational semantics of events.

$$\frac{expr : \texttt{activate behaviour}\ x\ \texttt{as}\ b(\overline{xexpr}^{*}) \quad expr \in C <: \texttt{Agent}}{b\ x = \texttt{new}\ b(\overline{xexpr}^{*});\ \texttt{this.addBehaviour}(x);} \tag{13}$$

$$\frac{expr : \texttt{activate behaviour}\ x\ \texttt{as}\ b(\overline{xexpr}^{*}) \quad expr \in C <: \texttt{Behaviour}}{b\ x = \texttt{new}\ b(\overline{xexpr}^{*});\ \texttt{theAgent.addBehaviour}(x);} \tag{14}$$

$$\frac{expr : \texttt{performative is}\ P \quad expr \in wexpr}{\texttt{MessageTemplate.matchPerformative}(P);} \tag{15}$$

$$\frac{w_1\ \texttt{or}\ w_2 \in wexpr}{\texttt{MessageTemplate.or}(w_1, w_2);} \quad \frac{w_1\ \texttt{and}\ w_2 \in wexpr}{\texttt{MessageTemplate.and}(w_1, w_2);} \quad \frac{\texttt{not}\ w \in wexpr}{\texttt{MessageTemplate.not}(w);} \tag{16}$$

Fig. 8. Rules that specify the operational semantics of relevant expressions.

contains the Java translation of the expressions contained in the `do` block, and the `run` method contains the usual pattern for message reception, as documented in virtually all teaching material on JADE (see, e.g., [31]). Finally, the `action` method of the behaviour runs in sequence all behaviour event handlers, which would typically check their condition and return immediately.

In Figure 8, the semantics of some interesting expressions is shown. Expressions are directly translated into Java code that uses the API of JADE. For example, the `activate-behaviour` construct declares a new object x of type b, and it adds the object to the list of behaviours of the agent by means of `addBehaviour`, which is a method of class `Agent`. Rules (13) and (14) show the activation of a behaviour in two cases, i.e., the activation inside an agent and inside another behaviour. Rules (15) and (16) show the translation of a `when` expression into a `MessageTemplate`.

V. Conclusions

This paper presented an overview of an operational semantics for the JADEL programming language. First, the syntax that JADEL provides for its main abstractions is shown and discussed. Then, relevant lookup functions and operational semantics rules are provided to offer an outlook on the complete operational semantics. In detail, JADEL agents and behaviours are mapped into Java classes using the lookup functions of FJ, a minimalistic subset of Java equipped with an operational semantics. In addition, new auxiliary functions are defined to treat the agent-oriented features of JADEL, which are obviously not part of FJ. Finally, relevant rules that formalize the operational semantics of JADEL expressions in terms of Java statements are presented. The major contribution of this work is to formalize the mapping between JADEL and Java with JADE, and to provide useful guidelines for code generation. Actually, the proposed operational semantics is the core of the current implementation of the JADEL compiler.

The current implementation of JADEL is in use to experiment on the new possibilities that the features of smart devices offer to agents. In particular, JADEL agents are given indoor localization capabilities [32], [33], in known environments but without a dedicated infrastructure, to experiment on location-aware games [10]. Moreover, JADEL has been used to experiment on the use of agent technologies to support effective collaborations in synergy with social networks [34]. All such works emphasized the effectiveness of JADEL in the implementation of agents that interact following complex protocols. In particular, the implementation of agents for experiments suggested interesting improvements of the language in the direction of incorporating support for declarative programming, as typically expected from an agent-oriented programming language (see, e.g., [35] for a recent discussion on the subject).

REFERENCES

[1] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi, "JADE – A Java agent development framework," in *Multi-Agent Programming: Languages, Platforms and Applications*, R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, Eds. Springer, 2005, pp. 125–147.

[2] F. Bergenti, "An introduction to the JADEL programming language," in *Procs. IEEE* 26^{th} *Int'l Conf. Tools with Artificial Intelligence (ICTAI)*. IEEE Press, 2014, pp. 974–978.

[3] F. Bergenti, E. Iotti, S. Monica, and A. Poggi, "A comparison between asynchronous backtracking pseudocode and its jadel implementation," in *Procs. of the* 9^{th} *Int'l Conference on Agents and Artificial Intelligence (ICAART)*, ser. ScitePress, vol. 2, 2017, pp. 250–258.

[4] K. Kravari and N. Bassiliades, "A survey of agent platforms," *Journal of Artificial Societies and Social Simulation*, vol. 18, no. 1, p. 11, 2015.

[5] F. Bergenti, "A discussion of two major benefits of using agents in software development," in *Third International Workshop on Engineering Societies in the Agents World (ESAW 2002)*, 2002, pp. 1–12.

[6] F. Bergenti, G. Caire, and D. Gotta, "Interactive workflows with WADE," in *Procs. of the* 21^{st} *IEEE International Conference on Collaboration Technologies and Infrastructures (WETICE 2012)*. IEEE, 2012, pp. 10–15.

[7] G. Caire, D. Gotta, and M. Banzi, "WADE: A software platform to develop mission critical applications exploiting agents and workflows," in *Procs. of the* 7^{th} *International Joint Conference on Autonomous Agents and Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 29–36.

[8] F. Bergenti, G. Caire, and 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.

[9] F. Bergenti, G. Caire, and D. Gotta, "Agent-based social gaming with AMUSE," in *Procs.* 5^{th} *Int'l Conf. Ambient Systems, Networks and Technologies (ANT 2014) and* 4^{th} *Int'l Conf. Sustainable Energy Information Technology (SEIT 2014)*, ser. Procedia Computer Science. Elsevier, 2014, pp. 914–919.

[10] F. Bergenti and S. Monica, "Location-aware social gaming with AMUSE," in *Advances in Practical Applications of Scalable Multi-agent Systems. The PAAMS Collection:* 14^{th} *International Conference, PAAMS 2016*, Y. Demazeau, T. Ito, J. Bajo, and M. J. Escalona, Eds. Springer International Publishing, 2016, pp. 36–47.

[11] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[12] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[13] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[14] F. Bergenti, E. Iotti, and 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, 2016, pp. 213–224.

[15] F. Bergenti, G. Rimassa, M. Somacher, and L. M. Botelho, "A FIPA compliant goal delegation protocol," in *Communication in Multiagent Systems: Agent Communication Languages and Conversation Policies*, M.-P. Huget, Ed. Springer, 2003, pp. 223–238.

[16] F. Bergenti, E. Iotti, S. Monica, and A. Poggi, "A case study of the JADEL programming language," in *Proceedings* 17^{th} *Workshop Dagli Oggetti agli Agenti* (WOA 2016), ser. CEUR Workshop Proceedings, vol. 1664. RWTH Aachen, 2016, pp. 85–90.

[17] F. Bergenti, E. Iotti, S. Monica, and A. Poggi, "Interaction protocols in the JADEL programming language," in *Procs.* 6^{th} *Int'l Workshop Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016)*. ACM Press, 2016, pp. 11–20.

[18] M. Yokoo and K. Hirayama, "Algorithms for distributed constraint satisfaction: A review," *Autonomous Agents and Multi-Agent Systems*, vol. 3, no. 2, pp. 185–207, 2000.

[19] S. Cagnoni, F. Bergenti, M. Mordonini, and G. Adorni, "Evolving binary classifiers through parallel computation of multiple fitness cases," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 35, no. 3, pp. 548–555, 2005.

[20] S. Monica and F. Bergenti, "A stochastic model of self-stabilizing cellular automata for consensus formation," in *Proceedings of* 15^{th} *Workshop Dagli Oggetti agli Agenti* (WOA 2014), ser. CEUR Workshop Proceedings, vol. 1260. RWTH Aachen, 2014.

[21] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: A minimal core calculus for Java and GJ," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 3, pp. 396–450, 2001.

[22] A. S. Rao, "AgentSpeak (L): BDI agents speak out in a logical computable language," *Agents Breaking Away*, pp. 42–55, 1996.

[23] M. Wooldridge, "A knowledge-theoretic semantics for concurrent MetateM," *Intelligent Agents III Agent Theories, Architectures, and Languages*, pp. 357–374, 1997.

[24] M. Challenger, M. Mernik, G. Kardas, and T. Kosar, "Declarative specifications for the development of multi-agent systems," *Computer Standards & Interfaces*, vol. 43, pp. 91–115, 2016.

[25] M. Challenger, S. Demirkol, S. Getir, M. Mernik, G. Kardas, and T. Kosar, "On the use of a domain-specific modeling language in the development of multiagent systems," *Engineering Applications of Artificial Intelligence*, vol. 28, pp. 111–141, 2014.

[26] M. Baldoni, C. Baroglio, and F. Capuzzimati, "Typing multi-agent systems via commitments," in *Engineering Multi-Agent Systems: Second International Workshop, EMAS 2014*, F. Dalpiaz, J. Dix, and M. B. van Riemsdijk, Eds. Springer, 2014, pp. 388–405.

[27] F. Bergenti, E. Iotti, and A. Poggi, "Outline of a formalization of JADE multi-agents system," in *Proceedings* 16^{th} *Workshop Dagli Oggetti agli Agenti* (WOA 2015), ser. CEUR Workshop Proceedings, vol. 1382. RWTH Aachen, 2015.

[28] F. Bergenti, E. Iotti, and A. Poggi, "An outline of the use of transition systems to formalize JADE agents and multi-agent systems," *Intelligenza Artificiale*, vol. 9, no. 2, pp. 149–161, 2015.

[29] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in *Procs. ACM Int'l Conf. Object Oriented Programming Systems Languages and Applications companion (OOPSLA 2010)*. ACM, 2010, pp. 307–309.

[30] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus, "Xbase: Implementing domain-specific languages for Java," in *Procs.* 11^{th} *Int'l Conf. Generative Programming and Component Engineering (GPCE 2012)*. ACM Press, 2012, pp. 112–121.

[31] F. Bellifemine, G. Caire, and D. Greenwood, *Developing multi-agent systems with JADE*, ser. Wiley Series in Agent Technology. John Wiley & Sons, 2007.

[32] S. Monica and F. Bergenti, "Location-aware JADE agents in indoor scenarios," in *Proceedings of* 16^{th} *Workshop Dagli Oggetti agli Agenti* (WOA 2015), ser. CEUR Workshop Proceedings, vol. 1382. RWTH Aachen, 2015, pp. 103–108.

[33] S. Monica and 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*. Springer International Publishing, 2016, pp. 111–123.

[34] F. Bergenti, E. Franchi, and A. Poggi, "Agent-based social networks for enterprise collaboration," in 20^{th} *IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2011, pp. 25–28.

[35] L. Fichera, F. Messina, G. Pappalardo, and C. Santoro, "A Python framework for programming autonomous robots using a declarative approach," *Science of Computer Programming*, vol. 139, pp. 36–55, 2017.