Outline of a Formalization of JADE Multi-Agent Systems

Federico Bergenti
Dipartimento di Matematica e Informatica
Università degli Studi di Parma
Parco Area delle Scienze 53/A, 43124 Parma, Italy
Email: federico.bergenti@unipr.it

Eleonora Iotti, Agostino Poggi
Dipartimento di Ingegneria dell'Informazione
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 proposes a formalization of JADE agents and multi-agent systems based on transition systems. The first section introduces the aims and scope of the research and it focuses the content of the paper. The second section enumerates the abstractions and the structures used in the formalization. Successively, third section presents the formal semantics of the parts of a JADE-based source code that are involved in the management of *(i)* the life cycle of agents and *(ii)* the behaviours of agents. Fourth section shows a very simple JADE agent and it exemplifies the use of the proposed transition system. Finally, a brief recapitulation of the work concludes the paper.**

I. Introduction

JADE [1] is today one of the most widely used tools for the development of multi-agent systems both in research and in the industry. It is a core component of a complex software system that helps managing one of the most penetrating telecommunication networks in Europe, serving millions of consumers daily [2]. It has been recently enhanced to support smart appliances [3], and it is the base of a recent initiative intended to revitalize the use of software agents to support social activities [4], both cooperative (see, e.g., [5], [6]) and competitive. Besides this, JADE was initially conceived primarily as a practical tool to help researchers experimenting with FIPA technology (www.fipa.org). JADE was recognized by FIPA community as the tool that most accurately implemented FIPA specifications, and it was often used for validating new specifications and for assessing the conformance of other tools to FIPA guidelines and specifications. Moreover, it was often selected to experiment potential enhancements to FIPA specifications (see, e.g., [7]), and to gather the interest of FIPA members on common projects (see, e.g., [8]). Such a close relationship with an official standardization body forced JADE architects to let the design of JADE APIs open, so that third-party developers could adopt JADE—and FIPA together with it—with minimal restrictions. For example, the agent model that JADE provides was intentionally left simple and not formally specified so that developers were not forced to adopt a specific agent model just because they wanted their agents to be FIPA compliant.

After more than 15 years of JADE development and use, this paper first proposes a formal semantics of JADE agents and multi-agent systems to support reasoning on JADE-based software systems. The proposed semantics is based on transition systems [9] and it closely follows the intended meaning of JADE APIs to ensure that properly written JADE agents can be reviewed in terms of the proposed semantics. No refinement of the JADE agent model is proposed and the semantics is ground on the semantics of Java classes. We always assume the availability of an underlying Java transition system and we refrain from formalizing it because it is a critical topic out of the scope of this paper (see, e.g., [10]).

For the sake of brevity, only an outline of the proposed semantics is presented in this paper, and interested readers are invited to consult an upcoming paper that provides a complete description of the formalization.

II. Definitions and Terminology

The proposed formalization considers only five main entities that collectively describe a JADE multi-agent system. The first of such entities is the multi-agent system itself—the MAS—which is seen as a sort of environment where agents live. Such an environment is subject to internal and external events that may modify its state and the state of each agent belonging to the MAS. Also, a MAS accounts for all entities that we need to formalize the JADE system: within a MAS we are able to see all agents and their respective behaviours. The second entity that we consider is the agent. An agent is a complex entity that has a state, detailed in Section II-C, and a list of behaviours. The behaviour is the third entity that we consider: it has a state, a type and an associated action. Actions are not entities in the proposed formalization, and they are always associated to behaviours. Actually, we may think of an action as a list of activites that an agent performs when it decides to activate the behaviour that encapsulates the action. The type of the behaviour changes the way the agent performs the associated action, e.g., a cyclic behaviour permits the agent to perform the action cyclically; conversely, a one-shot behaviour in meant to have the action performed only once. Each agent (respectively, behaviour) belongs to one agent class (respectively, behaviour class), and such classes are the last two of the five entities that we consider. An agent class is defined as a group of agents that have behaviours belonging to the same classes, and a behaviour class is a group of behaviours that have the same type and the same action. It is worth noting that the word class was chosen to identify the means that JADE provides to group agents and behaviours by specific properties, i.e., Java classes. This choice implicitly types agents, but this work does not address this issue and related issues (see, e.g., [11]).

A. Constants

In order to give a formal definition of the aforementioned five entities, we choose a first-order language whose alphabet consists of a finite set of constants, predicate symbols, function symbols, and the usual logical symbols (i.e., variables, connectives, and quantifiers). We let $\mathcal{C}$ be the finite set of constants and $\mathcal{V}$ the countably infinite set of variables.

In order to account for the life cycle state of an agent, we define specific constants:`initiated`, `deleted`, `active`, `suspended`, and `waiting`. Similarly, the life cycle state of a behaviour is described by constants `initiated`, `active`, `done`, and `blocked`. We call S_A the finite set of the agent life cycle state constants, and S_B the finite set of behaviour life cycle state constants. Each behaviour needs a type and a description of its internal state: we call T and S the sets of such constants, respectively. Moreover, we also need a finite set of constants that describes the state of the environment where agents live, which we call W.

In summary, the set of constants that we allow for the first-order language used to describe the proposed semantics is composed of the union of all aforementioned sets:

$$\mathcal{C} = S_A \cup S_B \cup T \cup S \cup W. \tag{1}$$

B. Variables and identifiers

In addition to constants and variables, we define a set of identifiers called I. Every time an entity is created, it gets a name from such a set. Each agent class and behaviour class is associated with a specific identifier in the set I. Therefore we need a semantic structure that associates the *name* of the class to the class itself. We call $\mathcal{F}_A$ the set of all agents classes and $\mathcal{F}_B$ the set of all behaviour classes. Such semantic structures are called *semantic contexts* of the classes, and they are

$$\rho_A : I \to \mathcal{F}_A, \tag{2}$$
$$\rho_B : I \to \mathcal{F}_B. \tag{3}$$

In JADE, each agent has a name and a global unique identifier (called *AID*, for agent identifier), which contains its local name and its platform name. So we can consider the AID as a structured identifier defined a function

$$\mathrm{aid} : \mathcal{V} \to I \tag{4}$$

that maps a variable a representing an agent to its AID. Moreover, each behaviour has a name, which is its unique identifier. We can access such an identifier through the function

$$\mathrm{bid} : \mathcal{V} \to I. \tag{5}$$

Each agent (respectively, behaviour) needs to be instantiated and maintained in a semantic structure, here generically called *heap*, which associates a concrete reference (i.e., an identifier) to the representation of the agent (respectively, behaviour). We call A the set of all agents and B the set of all behaviours, and we define a function α that accesses the state of an agent as

$$\alpha : I \to A. \tag{6}$$

Similarly, we define the function β that accesses the state of a behaviour as

$$\beta : I \to B. \tag{7}$$

The composition $\alpha \circ \mathrm{aid} : \mathcal{V} \to A$ associates an agent to a variable in the first-order language, while the composition $\beta \circ \mathrm{bid} : \mathcal{V} \to B$ does the same with respect to a behaviour. We may write $\alpha(a)$ instead of $\alpha(\mathrm{aid}(a))$ with a slight abuse of notation when the intended meaning is evident from the context.

C. Events

A MAS is constantly subject to two types of events: internal and external. Each agent is fed with a sub-sequence of such events in its life. Some of such events are particularly important because they can change the state of the MAS (these are the external events), and/or the state of agents and respective behaviours (these are the internal events). The set of such notable events that JADE manages, which we call E, contains the following events:

- $\mathrm{Create}(a, F)$ denotes the creation of the agent a of class F;
- $\mathrm{Kill}(a)$ denotes the destruction of the agent a;
- $\mathrm{Wait}(a)$ denotes the state change of the agent a to the state `waiting`;
- $\mathrm{Wake}(a)$ denotes the state change of the agent a from `waiting` to `active`;
- $\mathrm{Suspend}(a)$ denotes the state change of the agent a to the state `suspended`;
- $\mathrm{Activate}(a)$ denotes the state change of the agent a from `suspended` to the state it had before becoming suspended;
- $\mathrm{Create}(a, b, F)$ denotes the creation of the behaviour b of class F and agent a;
- $\mathrm{Block}(b)$ denotes the state change from `active` to `blocked` for the behaviour b;
- $\mathrm{Restart}(b)$ denotes the state change from `blocked` to `active` for the behaviour b;
- $\mathrm{Start(MAS)}$ denotes the first event that occurs in the MAS; and
- $\mathrm{End(MAS)}$ denotes the event that marks the termination of the MAS (with all agents and behaviours); it is the last event that occurs in a computation of the MAS.

The events in E that denotes changes in the state of an agent are related to each other as shown in Figure II-C (left). Similarly, Figure II-C (right) shows the finite state machine that represents the state changes of a behaviour.

In order to analyze the life cycle of agents and behaviours, we need to consider which event occurs in single steps of the computation: if $e \in E$ and $t \in \mathbb{N}$, the pair $\langle t, e \rangle$ means that the event e occurred at step t.

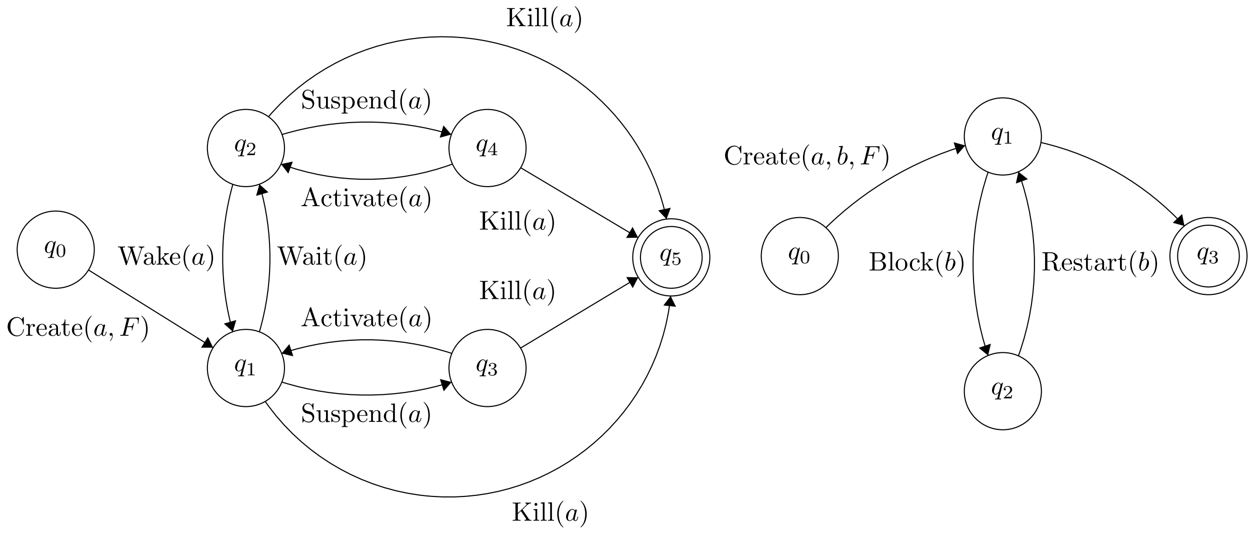

Figure 1. Agent (left) and behaviour (right) life cycle events and their relationships. For agents: q_0 = `initiated`, q_1 = `active`, q_2 = `waiting`, q_3 = `suspended_from_active`, q_4 = `suspended_from_waiting` and q_5 = `deleted`; for behaviours q_0 = `initiated`, q_1 = `active`, q_2 = `blocked` and q_3 = `done`.

D. Structures

An agent class is a pair $F_a = \langle L_B, \varphi \rangle$, where L_B is a list of possible behaviours and φ a function that maps the names of its methods (in the set I of identifiers) to the associated blocks of code, i.e., sequences of statements written in Java with JADE.

Similarly, a behaviour class is a pair $F_b = \langle t, \psi \rangle$, where $t \in T \subseteq \mathcal{C}$ fixes the type of the behaviour in the class (e.g., *cyclic* or *one-shot*), and ψ is a function that associates specific names to the related blocks of code.

In JADE, an agent is an entity that belongs to a particular class and that has a list of behaviours. So, an agent is completely described by the name of its class and by its state, as follows.

Definition 1 (Agent): An *agent* is a tuple

$$\langle \mathtt{F}, s_a, s_w, L_b, L_e \rangle \in A \tag{8}$$

where $\mathtt{F} \in I$ is the identifier of the class, $s_a \in S_A$ is the state of the agent, $s_w \in W$ the state of the environment seen by the agent, L_b is a finite list of behaviours and L_e a finite list of events.

Similarly, we can define a behaviour as a specific entity which belongs to a class, as follows.

Definition 2 (Behaviour): A *behaviour* is a tuple

$$\langle \mathtt{F}, \mathtt{myAgent}, s_b, s_w, s \rangle \in B \tag{9}$$

where $\mathtt{F} \in I$ is the identifier of the class, $\mathtt{myAgent} \in I$ is the unique identifier of the associated agent, $s_b \in S_B$ is the state of the behaviour, $s_w \in W$ is the state of the environment seen by the behaviour and $s \in S$ is the internal state of the behaviour. We call B the set of all behaviours.

In JADE, each agent lives within a MAS and therefore, for every time step T we can consider a global list of events L_E that occurred to all agents of that MAS since its activation, i.e., since event $\mathrm{Start(MAS)}$.

Definition 3 (MAS): A *MAS* is defined by a tuple

$$\mathrm{MAS} = \langle A', B', L_E \rangle_T \tag{10}$$

where $A' \subseteq A$ is a finite set of agents, $B' \subseteq B$ is a finite set of behaviours, L_E is the list of events occurred in this MAS up to time step $T \in \mathbb{N}$.

In addition, to support the semantics described in next section, we define $\mathcal{S}_b$, a selector for the behaviours of an agent, i.e., the *behaviour scheduler*. Such a function computes the current behaviour to execute, from the list L_b of an agent a as

$$\mathcal{S}_b : A' \to B'. \tag{11}$$

III. Semantics

The semantics of the life cycle of agents, given the events described in Section II-C, is described in the form of a *labelled transition system* (see, e.g., [9]). The proposed transition system, called *MAS transition system*, is a pair $\langle \Gamma, \to \rangle$ where Γ is a finite set of configurations and $\to$ is a binary relation on Γ. Once a MAS is fixed, the labels of such a transition system are the pairs time-event defined above.

Definition 4 (Configuration): A *configuration* in the transition system is described by a tuple

$$\langle C, \sigma, \alpha, \beta \rangle \in \Gamma \tag{12}$$

where C is a command, σ represents the state (stores and environments) of the underlying Java system, α and β are the heaps that represent the state of all agents and of all behaviours, respectively.

For the sake of clarity and when no ambiguity can arise, we may not enumerate all four elements of a configuration in the description of transition rules. Moreover, we use the symbol ε for a configuration with no commands. The abstract syntax used for a command (or statement) C is the Java syntax, as defined, e.g., in [10].

Each transition refers to an environment, or semantic context. In this case, we use the class contexts defined in Section II-D. So, for example, a transition may be written as follows

$$\rho_A, \rho_B \vdash_N \langle C, \sigma, \alpha, \beta\rangle \xrightarrow{\langle t,e\rangle} \langle C', \sigma', \alpha', \beta'\rangle. \qquad (13)$$

where ρ_A and ρ_B are the class contexts and $N \subseteq I \cup \mathcal{V}$ is a finite set of *names*, i.e., identifiers or variables used in the transition.

A. Semantics of commands with events

We need a formal semantics to manage the events and the state changes that occur in the life cycle of an agent. To this extent, we use a subsystem of the main MAS transition system. Such a subsystem has the same configuration of the MAS system, but the transition relation is called $\rightarrow_{\text{com}}$. The execution of such a system may be interrupted by an event that may cause a state change: this is represented by taking the label on the transition relation that indicates the current execution step and by matching it with the event just occurred. Notably, at some execution step no event occurs, and we indicate them with $\langle t, \varpi\rangle$.

Let us consider a fixed MAS $\langle A', B', L_E\rangle_T$. A list of statements, indicated with $C_1; C_2$, is computed as usual from left to right taking care of the propagation of the current event, as shown by the rule

$$\frac{\langle C_1, \sigma, \alpha, \beta\rangle \xrightarrow{\langle t,e\rangle}_{\text{com}} \langle \varepsilon, \sigma', \alpha', \beta'\rangle}{\langle C_1; C_2, \sigma, \alpha, \beta\rangle \xrightarrow{\langle t,e\rangle}_{\text{com}} \langle C_2, \sigma', \alpha', \beta'\rangle}. \qquad (14)$$

When no event occurs, as we can see in the rule (15), the statement C is computed by means of the underlying Java semantics system, which we call $\rightarrow_{\text{jstmt}}$

$$\frac{\langle C, \sigma\rangle \rightarrow_{\text{jstmt}} \langle \varepsilon, \sigma'\rangle}{\langle C, \sigma\rangle \xrightarrow{\langle t,\varpi\rangle}_{\text{com}} \langle \varepsilon, \sigma'\rangle}. \qquad (15)$$

We work under the reasonable assumption that such a subsystem may change only the state σ and we do not consider changes in all stores and environments of Java.

We then define specific rules to update the state of an agent. When an event e related to the agent $a = \langle \mathtt{F}, s_a, s_w, L_b, L_e\rangle$ occurs, s_a is changed to a new state, and e is stored in the list of events L_e. Similarly, we define rules for the behaviour state changes. In this case, there is no need for a list of events inside the behaviour entity, as shown in Figure II-C. For example, the following rule describes the state change from `active` to `blocked`

$$\frac{e = \text{Block}(b) \quad \beta(b) = \langle \mathtt{F}, \mathtt{active}, s_w, s\rangle}{\langle C, \beta\rangle \xrightarrow{\langle t,e\rangle}_{\text{com}} \langle C, \beta[\langle \mathtt{F}, \mathtt{blocked}, s_w, s\rangle / b]\rangle}. \qquad (16)$$

There is no explicit rule for the change from `active` to `done` because the latter is reached only after the execution of the action for a one-shot behaviour, and cyclic behaviours never change to `done`. Obviously the event End(MAS) stops all behaviours and agents in a MAS, forcing them to reach their respective final states.

B. Agent life cycle

We describe the life cycle of a generic agent, instantiated in the MAS $\langle A', B', L_E\rangle_T$, starting from its creation. When a new agent $a \in A$ of class $\mathcal{F}_a = \langle L_B, \varphi\rangle$ is created, it is memorized in a location `a` of the heap α and the function `setup` of its class is executed. We assume that the code of the method `setup` is accessible from the function φ of the agent class

$$\varphi(\mathtt{setup}) = \{\mathtt{B}\}. \qquad (17)$$

In this case, the MAS system has the transition

$$\rho_A \vdash \langle \varepsilon, \alpha\rangle \xrightarrow{\langle t,e\rangle} \langle \mathtt{a.setup()}\{\mathtt{B}\}, \alpha'\rangle \qquad (18)$$

where $\alpha' = \alpha[\langle \mathtt{F}, \mathtt{initiated}, \omega, \mathtt{[]}, [\text{Create}(a, F)]\rangle / \mathtt{a}]$. Note that the agent class context is crucial within this transition because it gives us the necessary information regarding the `setup` method.

The block `B` of the agent `setup` is computed by the commands transition system, as follows

$$\frac{\rho_A \vdash_{\mathtt{a}} \langle \mathtt{B}, \sigma, \alpha, \beta\rangle \xrightarrow{\langle t,e\rangle}_{\text{com}} \langle \varepsilon, \sigma', \alpha', \beta'\rangle \quad \dots}{\rho_A \vdash \langle \mathtt{a.setup()}\{\mathtt{B}\}, \sigma, \alpha, \beta\rangle \xrightarrow{\langle t,e\rangle} \langle \varepsilon, \sigma', \alpha'', \beta'\rangle} \qquad (19)$$

where $\alpha'' = \alpha'[\langle \mathtt{F}, \mathtt{active}, s_w, L_b, L_e\rangle / \mathtt{a}]$.

During `setup`, agent a can add and/or remove behaviours from its list L_b, through the following statements

`addBehaviour(b),  removeBehaviour(b).`

It is worth noting that such statements can be used also inside the action of a behaviour, and not only in the `setup` method, to change the list of behaviours dynamically. Finally, after `setup`, the agent enters the `active` state and it becomes ready to select a behaviour in its list. The life of an agent continues by performing actions of behaviours until the event Kill(a) occurs. When an agent a is killed, it computes the function `takeDown`. Only after the execution of `takeDown` the agent a is removed from the heap by assigning $\alpha(\mathtt{a}) = \varpi$.

C. Behaviour actions

If $\alpha(\mathtt{a}) = a = \langle \mathtt{F}', s_a, s'_w, L_b, L_e\rangle \in A'$ is an instantiated agent and $\beta(\mathtt{b}) = b = \langle \mathtt{F}, \mathtt{a}, \mathtt{active}, s_w, s\rangle \in B'$ is a behaviour that appears in the list of the agent, i.e., $b \in L_b$, then the behaviour may be selected by the behaviour scheduler of the agent to perform the relative action, i.e., $\mathcal{S}_b(a) = b$. The transition of the main system then calls the `action` method of the class of b

$$\rho_B \vdash \langle \varepsilon, \sigma, \alpha, \beta\rangle \rightarrow \langle \mathtt{b.action()}\{\mathtt{C}\}, \sigma, \alpha, \beta\rangle.$$

Just like for the method `setup` of the agent, the block of code `C` of `action` is executed by the subsystem $\rightarrow_{\text{com}}$.

A one-shot behaviour terminates after a single execution of its action, and it is removed from the list of behaviours of the agent, through the function `removeBehaviour`. So, if the agent is still alive, the behaviour scheduler chooses another behaviour. On the contrary, if the behaviour b is cyclic, then its action is performed cyclically and it is never automatically removed from the list of behaviours of the agent.

$$\frac{\rho_A, \rho_B \vdash_{\mathrm{a,b}} \langle \texttt{C; removeBehaviour(b)}, \sigma, \alpha, \beta \rangle \xrightarrow{\langle t,e \rangle}_{\mathrm{com}} \langle \varepsilon, \sigma', \alpha', \beta' \rangle \quad \dots \quad \beta'(\texttt{b}) = \langle \texttt{F}, \texttt{a}, \texttt{active}, s_w, s \rangle}{\rho_B \vdash \langle \texttt{b.action()\{C\}}, \sigma, \alpha, \beta \rangle \xrightarrow{\langle t,e \rangle} \langle \varepsilon, \sigma', \alpha', \beta'[\langle \texttt{F}, \texttt{a}, \texttt{done}, s_w, s \rangle / \texttt{b}] \rangle} \tag{20}$$

$$\frac{\rho_A, \rho_B \vdash_{\mathrm{a,b}} \langle \texttt{C; removeBehaviour(b)}, \sigma, \alpha, \beta \rangle \xrightarrow{\langle t,e \rangle}_{\mathrm{com}} \langle \varepsilon, \sigma', \alpha', \beta' \rangle \quad \dots \quad \beta'(\texttt{b}) = \langle \texttt{F}, \texttt{a}, \texttt{blocked}, s_w, s \rangle}{\rho_B \vdash \langle \texttt{b.action()\{C\}}, \sigma, \alpha, \beta \rangle \xrightarrow{\langle t,e \rangle} \langle \varepsilon, \sigma', \alpha', \beta' \rangle} \tag{21}$$

Figure 2. Rules governing the transition of a one-shot behaviour from the active to the blocked state. The behaviour scheduler of an agent can select only behaviours that are in the active state.

Note that a necessary condition for the scheduler $\mathcal{S}_b(a)$ to choose a behaviour is that it must be in the active state. A behaviour is normally in the active state, but it can reach the blocked state during the execution of its action. When this occurs, the execution continues up to the natural termination of the action and the blocked behaviour is not removed from the list of behaviours. Anyway, the scheduler cannot choose it until it returns to the active state, as detailed, for one-shot behaviours, in rules (20) and (21), summarized in Figure 2.

IV. A Didactic Example

This section discusses a simple example of JADE agent and shows its semantics according to the transition system outlined in previous sections. The simple agent has the following source code.

```java
import jade.core.Agent;
import jade.core.behaviours.OneShotBehaviour;

public class HelloAgent extends Agent {
  protected void setup() {
    Greetings greetings = new Greetings(this);

    addBehaviour(greetings);
  }

  private static class Greetings extends
      OneShotBehaviour {
    public Greetings(Agent a) {
      super(a);
    }

    public void action() {
      System.out.println("Hello world");
    }
  }
}
```

Class `HelloAgent` defines an agent class and its static inner class a behaviour class. This is formalized by setting the pairs $\langle L_B, \varphi \rangle$ and $\langle t, \psi \rangle$ using the information contained in the source code. Then, the context is initialized to create a reference between the identifier of classes and their relative structures, as follows.

$$\rho_A(\texttt{HelloAgent}) = \langle L_B, \varphi \rangle$$
$$\rho_B(\texttt{Greetings}) = \langle \texttt{one-shot}, \psi \rangle$$
$$L_B = [\texttt{Greetings}]$$

$\varphi(\texttt{setup}) = \{$
```
    Greetings greetings =
        new Greetings(this);
    addBehaviour(greetings);
```
$\}$

$\psi(\texttt{action}) = \{$
```
    System.out.println("Hello world");
```
$\}$

Suppose that the MAS starts at execution step 0, and that the first event that occurs is the creation of agent *hello*, belonging to the class *HelloAgent*

$$\begin{gathered} e_0 = \langle 0, \mathrm{Start(MAS)} \rangle \quad \mathrm{MAS} = \langle \emptyset, \emptyset, [e_0] \rangle_T \\ \downarrow \\ e_1 = \langle 1, \mathrm{Create(hello, HelloAgent)} \rangle \\ A' = \{\mathrm{hello}\},\ L_E = [e_1, e_0]. \end{gathered}$$

Variable *hello* allows retrieving the information about the new agent, and we can access its AID by function aid(hello) $=$ `hello`, and control its state thanks to the heap α.

The event Create(hello, HelloAgent) activates the first transition

$$\rho_A \vdash \langle \varepsilon, \alpha \rangle \xrightarrow{e_1} \langle \texttt{hello.setup()}\{\texttt{B}\}, \alpha^i \rangle$$

where $\{\texttt{B}\}$ is the block of code obtained by $\varphi(\texttt{setup})$ and

$$\alpha^i = \alpha[\langle \texttt{HelloAgent}, \texttt{initiated}, \omega, \texttt{[]}, [e_1] \rangle \\ /\texttt{hello}].$$

Suppose that no event occurs at execution step 2: the main transition system enters in the subsystem $\rightarrow_{\mathrm{com}}$, as follows.

$$\begin{gathered} \rho_A \vdash_{\mathrm{hello}} \langle \texttt{Greetings greetings =} \\ \texttt{new Greetings()}, \sigma, \alpha^i, \beta \rangle \\ \downarrow e_3 \\ \langle \texttt{addBehaviour(greetings)}, \sigma^i, \alpha^i, \beta^i \rangle \\ \downarrow e_4 \\ \langle \varepsilon, \sigma^{ii}, \alpha^{ii}, \beta^i \rangle \end{gathered}$$

where σ^{ii} is obtained from a transition of the Java subsystem $\rightarrow_{\text{jstmt}}$, called at every execution step, and

$$\beta^{i} = \beta[\langle \texttt{Greetings}, \texttt{hello}, \texttt{active}, \omega, \omega\rangle \\ /\texttt{greetings}]$$

$$\alpha^{ii} = \alpha^{i}[\langle \texttt{HelloAgent}, \texttt{initiated}, \\ \omega, [\texttt{greetings}], [e_3, e_1]\rangle/\texttt{hello}].$$

This transition is necessary to apply rule (19)

$$\frac{\rho_A \vdash_{\text{hello}} \langle \texttt{B}, \sigma, \alpha^{i}, \beta\rangle \xrightarrow{e_2}_{\text{com}} \langle \varepsilon, \sigma^{ii}, \alpha^{ii}, \beta^{i}\rangle \quad \ldots}{\rho_A \vdash \langle \texttt{hello.setup()}\{\texttt{B}\}, \sigma, \alpha^{i}, \beta\rangle \xrightarrow{e_2} \langle \varepsilon, \sigma^{ii}, \alpha^{iii}, \beta^{i}\rangle}$$

where the state of the agent is eventually changed to active

$$\alpha^{iii} = \alpha^{ii}[\langle \texttt{HelloAgent}, \texttt{active}, \\ \omega, [\texttt{greetings}], [e_3, e_1]\rangle/\texttt{hello}].$$

Then, because agent *hello* is active, the scheduler can choose a behaviour to execute. The only behaviour in the list of behaviours of the agent is *greetings* so $\mathcal{S}_b(\text{hello}) = \text{greetings}$

$$\rho_B \vdash \langle \varepsilon, \sigma^{ii}, \alpha^{iii}, \beta^{i}\rangle \\ \downarrow e_5 \\ \langle \texttt{greetings.action()}\{\texttt{B}\}, \sigma^{ii}, \alpha^{iii}, \beta^{i}\rangle.$$

The block of code $\{\texttt{B}\}$ of the action, accessed through the function ψ, is executed in the $\rightarrow_{\text{com}}$ subsystem, as follows.

$$\rho_A, \rho_B \vdash_{\text{hello,greetings}} \\ \langle \texttt{System.out.println("Hello world")} \sigma^{ii}, \alpha^{iii}, \beta^{i}\rangle \\ \downarrow e_6 \\ \langle \varepsilon, \sigma^{iii}, \alpha^{iii}, \beta^{i}\rangle$$

where σ^{iii} is obtained by the Java transition system. According to rule (20), at the end of the computation of its action, the state of the behaviour is changed to done.

$$\frac{\rho_A, \rho_B \vdash_{\text{hello,greetings}} \langle \texttt{B}, \sigma^{ii}, \alpha^{iii}, \beta^{i}\rangle \\ \xrightarrow{e_6}_{\text{com}} \langle \varepsilon, \sigma^{iii}, \alpha^{iii}, \beta^{i}\rangle \quad \ldots}{\rho_A \vdash \langle \texttt{greetings.action()}\{\texttt{B}\}, \sigma^{ii}, \alpha^{iii}, \beta^{i}\rangle \\ \xrightarrow{e_6} \langle \varepsilon, \sigma^{iii}, \alpha^{iii}, \beta^{ii}\rangle}.$$

In fact:

$$\beta^{ii} = \beta^{i}[\langle \texttt{Greetings}, \texttt{hello}, \texttt{done}, \omega, \omega\rangle \\ /\texttt{greetings}]$$

Let us now suppose that when $T = 7$ the event End(MAS) occurs, thus producing the following transition

$$\rho_A, \rho_B \vdash \langle \varepsilon, \sigma^{iii}, \alpha^{iii}, \beta^{ii}\rangle \xrightarrow{e_7} \langle \texttt{E}, \sigma^{iii}, \alpha^{iii}, \beta^{ii}\rangle$$

where E is a special command which denotes the termination of the execution. The event occurred are summarized below.

$$e_2 = \langle 2, \varpi\rangle \\ \downarrow \\ e_3 = \langle 3, \text{Create}(\text{hello}, \text{greetings}, \text{Greetings})\rangle \\ \downarrow \\ e_4 = \langle 4, \varpi\rangle \\ \downarrow \\ e_5 = \langle 5, \varpi\rangle \\ \downarrow \\ e_6 = \langle 6, \varpi\rangle \\ \downarrow \\ e_7 = \langle 7, \text{End}(\text{MAS})\rangle$$

V. Conclusions

This paper provides an outline of the major ingredients of a semantics of JADE agents and multi-agent systems based on transition systems. The five main entities that we consider, namely the multi-agent system, agents and their classes, behaviours and their classes, are discussed. The paper provides an example of the transition system that can be obtained from a very simple JADE agent. The complete description of the semantics cannot fit the constraints of a workshop paper and interested readers are directed to an upcoming journal paper that also discusses the complete semantics of message passing.

References

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

[2] 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.

[3] F. Bergenti, G. Caire, and D. Gotta, "Agents on the move: JADE for Android devices," in *Procs. Workshop From Objects to Agents*, 2014.

[4] 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.

[5] F. Bergenti and A. Poggi, "Agent-based approach to manage negotiation protocols in flexible CSCW systems," in *Procs. 4^{th} Int'l Conf. Autonomous Agents*, 2000, pp. 267–268.

[6] F. Bergenti, A. Poggi, and M. Somacher, "A collaborative platform for fixed and mobile networks," *Communications of the ACM*, vol. 45, no. 11, pp. 39–44, 2002.

[7] F. Bergenti and A. Poggi, "Ubiquitous information agents," *Int'l J. Cooperative Information Systems*, vol. 11, no. 34, pp. 231–244, 2002.

[8] F. Bergenti, A. Poggi, B. Burg, and G. Caire, "Deploying FIPA-compliant systems on handheld devices," *IEEE Internet Computing*, vol. 5, no. 4, pp. 20–25, 2001.

[9] G. D. Plotkin, "A Structural approach to Operational Semantics," *J. Log. Algebr. Program.*, vol. 60-61, pp. 17–139, 2004.

[10] J. Alves-Foss, *Formal syntax and semantics of Java*. Springer Science & Business Media, 1999, no. 1523.

[11] M. Baldoni, C. Baroglio, and F. Capuzzimati, "Typing multi-agent systems via commitments," *Post-Procs. 2^{nd} Int'l Workshop on Engineering Multi-Agent Systems (EMAS 2014), Revised Selected and Invited Papers*, pp. 388–405, 2014.