A Case Study of the JADEL Programming Language

Federico Bergenti*, Eleonora Iotti†, Stefania Monica* and Agostino Poggi†
* Dipartimento di Matematica e Informatica
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 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 presents and discusses a first complete example of the use of JADEL. JADEL is a novel agent-oriented domain-specific programming language built on top of JADE, the well-known agent platform which provides solid agent technology and several tools for the creation of agents and multi-agent systems. The purpose of JADEL is to make the development of JADE agents and multi-agent systems easier and clearer by means of specific abstractions and a lighter syntax. In order to understand, and properly assess, the actual advantages of using JADEL, a well-known JADE demo that uses JADE-specific features like ontologies and interaction protocols has been rewritten in JADEL. This paper first briefly presents the main features of JADEL, then it discusses the rewritten demo and compares it with original JADE code.**

I. Introduction

JADEL (*JADE Language*) [1] is a novel programming language that provides agent-oriented abstractions and domain-specific expressions to help the development of JADE multi-agent systems. JADE (*Java Agent DEvelopment framework*, jade.tilab.com) [2] is a software framework that permits to build complex and distributed multi-agent systems. A wide variety of extensions and APIs are provided with JADE and the documentation of JADE APIs is exhaustive and clear. JADE allows use of agent technology in various areas, such as smart emergency applications [3] and localization [4], [5]. JADE is currently maintained and in recent years some related projects have been developed. In particular, *WADE* [6], [7] (*Workflows and Agents Development Environment*) adds a lightweight workflow engine to JADE agents, thus supporting business process management, and *AMUSE* [8]–[10] (*Agent-based Multi-User Social Environment*) focuses on multi player Android-based online games. These are some of the reasons that make JADE one of the most comprehensive and hence most popular FIPA-compliant agent platform [11]. Although there is a wide and clear documentation of JADE APIs, new developers and students in the field of agents and multi-agent systems may have difficulties in approaching JADE. As a matter of fact, users of JADE have to deal with lots of complementary technical details that are sometimes perceived as difficult and confusing in terms of agent-oriented features. These difficulties are due to the fact that JADE has grown in complexity and now it has a steep learning curve.

JADE allows users to make appropriate design choices for their specific application domains, without forcing them to rely on a specific agent model. Such flexibility and adaptability makes it a very powerful instrument that can be applied to several domains. The disadvantage of such an approach is that the agent-oriented features have been used in many different ways, resulting in a loss of transparency of the agent model.

Finally, the *AOP* (*Agent-Oriented Programming*) paradigm is inherently different from the *OOP* (*Object-Oriented Programming*) one. As a matter of fact, agents are characterized by mental states and they exchange specific types of messages, responding to them in a truthful and consistent way. To this extent, agents are viewed as specialization of objects [12], [13]. For this reason, the management of agents and multi-agent systems often requires the use of specific languages, which focus on agent-oriented abstractions and provide particular constructs and structures. In order to match the AOP paradigm, JADE APIs offer facilities written in Java. This design choice was made in the early stage of JADE development and it presents several advantages: it makes possible to develop applications compliant with agent-oriented technologies and it ensures interoperability with other Java libraries and applications. Despite its advantages, at the present day, a Java-only approach is often perceived as a limitation, due to the rising of several valid alternatives and the increased popularity of domain-specific solutions tailored on the needs of the application and of its context. Such solutions are often *DSL* (*Domain-Specific Languages*) designed intentionally to deal with the chosen domain. These languages have a lighter syntax and help avoiding repetitive tasks, workaround and unclear technicalities.

In summary, the main problems in developing agent-based applications and multi-agent systems in JADE today can be listed as follows. First, the complexity of the framework requires not only expertise in management of distributed multi-agent systems but also a deep understanding of JADE mechanisms. Second, the lack of a fixed agent model sometimes

causes unclear or incorrect utilizations of the available agent technologies. Finally, some procedures and patterns become repetitive or verbose, hence not clear in terms of AOP, due to the gap between the AOP and OOP paradigms.

The solution proposed to address these problems is JADEL, which was first presented in [1], [14]. JADEL is an agent-oriented DSL which aims at reducing the complexity of JADE by providing a new syntax relying completely on agent abstractions. According to this syntax, constructs and expressions are introduced in order to write simple and clear agent programs. JADEL is developed with the Xtext framework (www.eclipse.org/Xtext), which ensures a tight integration with the JVM and provides a specific language that can be used as host language of the DSL. This specific language is called Xtend (www.eclipse.org/xtend) and it is a dialect of Java. An advantage of the use of Xtend is that JADEL programs generate Java source code which is easily readable and which runs as fast as the equivalent Java code. This feature makes also possible the embedding of JADE native code. As a consequence, the use of JADEL is quite straightforward for Java and JADE users. JADEL is still under development and an evaluation of its capabilities, usability and effectiveness is necessary. This paper presents a JADEL translation of the Meeting Scheduler demo, a well-known example of JADE that comes together with JADE distribution.

The structure of the paper is as follows. In Section II, JADEL main ideas and abstraction are presented, together with a brief explanation of its syntax. Then, Section III introduces the Meeting Scheduler example in JADEL and its comparison with the JADE implementation. Finally, Section IV concludes the paper with a discussion of the results and future developments and improvements.

II. The JADEL Programming Language in Brief

As previously explained, the scope of JADEL is to simplify the creation of JADE agents and multi-agent systems and to make the agent model clearly visible, avoiding technical and implementation details as much as possible. JADE APIs consist in a large amount of classes that have various purposes, such as the representation of core abstractions and the management of message passing among agents, from interaction protocols to agent communication languages. Hence, the choice of the primary features, among those provided by JADE classes, is fundamental in the development of a JADE DSL. In particular, two criteria for the selection of JADEL features are considered, namely *(i)* the significance of such features in agent creation and life-cycle and *(ii)* the FIPA compliance and the help they give to manage message passing. According to the previously listed criteria, JADEL core abstractions are the *agent*, the *behaviour* and the *communication ontology*. These abstraction are explained in detail in Section II-A.

Features of such entities are designed as domain-specific constructs and expressions, whose purpose is to shorten the most repetitive tasks and to focus on the agent life-cycle and behaviour action. Constructs are meant to handle events, e.g., the reception of a messages and creation or destruction of agents, while expressions are used, for instance, to create agent behaviours. In Section II-B such features are described in detail.

The above listed criteria must not be seen as a limitation. On the contrary, they represent a tentative of making JADEL code simple and readable, without detracting from its expressiveness. As a matter of fact, the underlying JADE platform and the interoperability with Java given by Xtext and Xtend ensure the possibility of embedding JADE native implementations.

A. Abstractions

In terms of syntax, all JADEL abstractions are declared in a similar way, namely with a keyword that identifies the abstraction, followed by the name of the entity and its relations with other declared entities. After the declaration, a block of code contains the description of its main features, by means of specific constructs.

The first entity that we discuss is the communication ontology. In detail, communication ontologies contain the definitions of *propositions*, *composite concepts* and *predicates*. From a theoretical point of view, propositions are first-order logic well-formed formulas. In JADEL they are declared simply by means of the keyword `proposition` followed by a name. Basic (or atomic) concepts are simple terms that can be used in JADEL, but they do not have to be declared, because they are provided directly by JADEL. Composite concepts are defined by means of other basic or composite concepts. According to JADEL syntax, composite concepts can be declared by using the keyword `concept` followed by the name of the concept and by a list of parameters. Such parameters identify the other concepts that compose the one that is being declared. Predicates have a similar syntax and their parameters can be basic or composite concepts. Predicates do not state a fact nor describe an entity, but they make relations among concepts, as in the semantics of logic predicates.

The second entity is the behaviour. In JADEL, behaviours are used to define actions that can be performed by agents and they can be `cyclic` or `oneshot`. The difference between cyclic and oneshot behaviours is that cyclic ones remain in the agent behaviours list for the entire life of an agent, while oneshot behaviours are removed after a single execution. Both of them can be associated with a specific agent class. A special consideration is given to those behaviours that are `role`s of an interaction protocol. Roles are behaviours defined by means of a FIPA protocol name and the role of the agent in such protocol. In the body of a role, each step of the protocol is handled individually as a particular event caused by the reception of different types of messages.

Finally, agents are the most important JADEL entities. In their declarations and setup, they put together all the ontologies and behaviours that will be used in their life cycles, thus making explicit relations among the various entities. Agents are multitask single-threaded entities that manage their actions by means of an internal scheduler. An agent life cycle is composed of: *(i)* a start-up phase, during which the agent initializes, *(ii)* a loop, where actions are performed, and *(iii)* a

final phase of take-down, in which the agent performs cleanup tasks. In JADEL, creation and destruction of an agent are events, so they are managed with specific constructs, whose syntax is coherent with the one of the behaviour events.

B. Domain-Specific Expressions

Particular expressions are used to manage events, message passing and domain-specific tasks, such as the behaviours registration in the agent list.

Events are managed by means of the keyword `on`, followed by the type of event. Events can be the creation and the destruction of an agent, the change of state of a participant in an interaction protocol, and the reception of a message. Notably, the JADEL construct `on − when − do` is used in a behaviour body to define an action. In detail, an action can be simple, i.e., it does not need an event to be triggered. In such a case, only the keyword `do` is used, followed by a block of code that describes the action. Otherwise, an action can be triggered by an event, and the keyword `on` specifies such event. The keyword `when` is optional and it can be used to set conditions on the event nature.

Communication among agents consists in message passing, which can be managed with ontologies and/or with interaction protocols. Interaction protocols provide a scenario with, at least, two different roles, *initiator* and *participant*, and establish the types of messages exchanged among agents that are in such roles. JADEL encourages the usage of such technologies by providing expressions that allow messages to be viewed as special data structures and by means of specific constructs that permit creating and sending messages, and extracting message contents. These expressions and constructs are exactly the same for each type of message, regardless of their content, which can be a string, a sequence of bytes, a concept, a proposition or a predicate.

The simple example below shows syntax and usage of some domain-specific expressions.

```
agent Ping {
  on create {
    var aidList = #[newAID('pong')]
    activate behaviour SendPing(aidList)
  }
}

agent Pong {
  on create {
    activate behaviour ReplyPong()
  }
}

oneshot behaviour SendPing(List<AID> aidList){
  do {
    send message {
      performative is INFORM
      receivers are aidList
      content is 'ping'
    }
  }
}

ontology PingPongOntology {
  proposition Ping
  proposition Pong
}

cyclic behaviour ReplyPong {
  on message msg
  when {
    content is 'ping'
  } do {
    send message {
      performative is INFORM
      receivers are #[msg.sender]
      ontology is PingPongOntology
      content is Pong
    }
  }
}
```

III. The Meeting Scheduler Demo

The Meeting Scheduler demo launches two (or more) agents, with their own GUI that consists in a calendar where the user can fix and view his/her appointments, invite known users to an appointment, and receive other users invitations. The agent is responsible for the management of the calendar, so the user does not have to bother about the overlap of two appointments. As a matter of fact, the agent schedules the appointments correctly by messaging with other agents.

The JADE source code of the demo consists in *(i)* a Meeting Scheduler agent, *(ii)* a set of classes that manage the GUI, *(iii)* three behaviours, and *(iv)* an ontology that defines two composite concepts. Regarding the behaviours, two of them are implementations of the initiator and responder roles of the FIPA Contract Net interaction protocol and the third deals with the cancellation of an appointment.

A. JADEL Implementation

The first entity implemented in JADEL is the communication ontology, with the concept `Person` and `Appointment`, whose sub-concepts are declared as parameters. The keyword `many` identifies a list. Declaration of `MSOntology` is self-contained and it does not need any other classes to define its two concepts, as in JADE.

```
package demo.meetingscheduler.ontologies

ontology MSOntology {
  concept Person(string name, aid AID, aid DFName)
  concept Appointment(aid inviter, string description,
    date startingOn, date endingWith, date FixedDate,
    many Person invitedPersons, many date possibleDates
  ) extends Cloneable
}
```

Then, the agent is declared as a `JadelGuiAgent`, an extension of the JADE class `GuiAgent` that provides some built-in functionalities, e.g., `log(String)` function, used to manage the logging for the agent. Classes that build the GUI are coded in Java, exactly as in the original demo. It is possible to refer to those classes into the agent body, thanks to the close integration that Xtext and Xtend provide with the underlying JVM.

First, some variables are declared in the `MeetingSchedulerAgent` entity. These will be translated into fields of the agent class. According to Xtend syntax, the keyword `val` denotes a private final field, while `var` stands for a mutable private field.

```
agent MeetingSchedulerAgent uses ontology MSOntology
    extends JadelGuiAgent {
  var String userName
  var Vector<AID> knownDF = new Vector()
  var HashMap<String, Person> knownPersons = newHashMap()
  var HashMap<String, Appointment> appointments =
      newHashMap()
  var mainFrame mf
  val int STARTTASKS = 2
  val int FIXAPPOINTMENT = 3
  val int REGISTERWITHDF = 4
  val int CANCELAPPOINTMENT = 5
  val int SEARCHWITHDF = 6
  val String NAME = 'AppointmentScheduling'
  val String TYPE = 'personal-agent'
  val String PROTOCOL = 'fipa-request fipa-Contract-Net'
```

Then, a setup method is used only to start the GUI, as follows.

```
  on create {
    var passwordDialog = new PasswordDialog(this,
      this.localName)
    passwordDialog.visible = true
  }
```

Finally, all needed methods of the agent are defined similarly to those in JADE implementation, with some difference regarding domain-specific expressions. In the following, two of such methods are shown.

```
  void startTasks(String name) {
    userName = name
    mf = new mainFrame(this, userName +
      '- Appointment Scheduler')
    mf.visible = true
    try {
      DFService.register(this,
        getDFAgentDescription(NAME, TYPE, MSOntology::NAME,
            userName, PROTOCOL))
      knownDF += defaultDF
      addKnownPerson(new Person(userName, AID, defaultDF))
    } catch (FIPAException e) {
      e.printStackTrace()
      mf.showErrorMessage(e.getMessage())
      log(e.getMessage(), Logger.WARNING)
    }
    take role myFipaContractNetResponderBehaviour(this)
    activate behaviour CancelAppointmentBehaviour(this)
  }


  void cancelAppointment(Date d) {
    var Appointment appointment = getMyAppointment(d)
    if (appointment == null) {
       mf.showErrorMessage('Nothing to cancel: no
           appointment was fixed on ' + d)
       log('Nothing to cancel: no appointment was fixed on
           ' + d, Logger.WARNING)
       return
    }
    if (!appointment.invitedPersons.empty) {
      var Action action = new Action()
      action.actor = AID
      action.action = appointment
      var receiversList = #[]
      for (p : appointment.invitedPersons)
        receiversList.add(p.AID)
      send message {
        performative is CANCEL
        ontology is MSOntology
        receivers are receiversList
        content is action
      }
    }
    removeMyAppointment(appointment)
  }
```

The roles are completely listed below and all the details required to define roles are given. Their declarations indicate the owner agent, protocol name and protocol role. As for the agent, a list of variables can be specified and the role creation is an event that can be handled as follows.

```
role myFipaContractNetInitiatorBehaviour(Appointment
    appointment, List<AID> group) for MeetingSchedulerAgent
     as FIPA_CONTRACT_NET:Initiator {
  var Appointment pendingApp
  var acceptableDates = newArrayList<Date>()
  var acceptedDates = newArrayList<Date>()

  on create {
    var action = new Action()
    action.setActor(theAgent.AID)
    action.setAction(appointment)

    create message cfpMsg {
      performative is CFP
      ontology is MSOntology
      receivers are group
      content is action
    }

    pendingApp = appointment
    println('Initiator msg: ' + cfpMsg)
  }
```

Roles are finite-state machine behaviours that change state when a new message with a certain performative is received. As explained in FIPA Contract Net specification, the *initiator* sends CFP messages to a group of agents and it has to handle their responses. In the following, negative responses are handled.

```
  on REFUSE msg {
    println('Initiator received Refuse: ' + msg.toString)
  }

  on NOT_UNDERSTOOD msg {
    println('Initiator handle NotUnderstood: ' + msg.
        toString)
  }

  on FAILURE msg {
    println('Initiator received Failure: ' + msg.toString)
  }
```

In case of acceptance of the CFP by the responder, initiator handles the propose message, as follows.

```
  on PROPOSE msg {
    var Calendar c = Calendar.instance

    for (d : pendingApp.possibleDates) {
      c.time = d
      acceptableDates.add(new Integer(c.get(c.DATE)))
    }

    extract proposal as Action from msg
    var appointment = proposal.action as Appointment

    for (d : appointment.possibleDates) {
      c.time = d
      var day = new Integer(c.get(c.DATE))
      if (acceptableDates.contains(day))
        acceptedDates.add(day)
    }

    acceptableDates = acceptedDates.clone
    if (!acceptableDates.empty) {
      var d = new Date()
      var int dateNumber =
        ((Integer)acceptableDates.get(0)).intValue()

      c.set(c.DATE, dateNumber)
      pendingApp.setFixedDate(c.time)

      var action = new Action()
      action.setActor(theAgent.AID())
      action.setAction(pendingApp)
```

```
      create message replyMsg {
        ontology is MSOntology
        performative is ACCEPT_PROPOSAL
        receivers are #[msg.sender]
        content is action
      }
    } else {
      create message replyMsg {
        performative is REJECT_PROPOSAL
        receivers are #[msg.sender]
        content is 'No date available'
      }
    }
  }
```

Finally, if the proposal is accepted and the responder succeeds in fixing the appointment, a final message that informs the initiator is sent. Here the handling of the reception of such message is shown.

```
  on INFORM msg {
    var Person p = theAgent.getPersonByAgentName(
      msg.sender)
    pendingApp.invitedPersons.clear
    if (p == null)
      p = new Person(msg.sender.name, null, null)
    pendingApp.getInvitedPersons().add(p)
    theAgent.addMyAppointment(pendingApp)
  }
```

Below, the source code of the *responder* behaviour is listed. In case the responder receives a CFP, it will decide to accept it or not.

```
role myFipaContractNetResponderBehaviour for
     MeetingSchedulerAgent as FIPA_CONTRACT_NET:Responder {
  on CFP msg {
    extract cfp as Action from msg
    var appointment = cfp.action as Appointment
    var proposal = appointment
    proposal.possibleDates.clear()
    for (d : proposal.possibleDates)
      if (theAgent.isFree())
        proposal.possibleDates.add(d)
    if (!proposal.possibleDates.empty) {
      var a = new Action()
      a.actor = theAgent.AID
      a.action = proposal
      create message reply {
        performative is PROPOSE
        ontology is MSOntology
        receivers are #[msg.sender]
        content is a
      }
    } else {
      create message reply {
        performative is REFUSE
        receivers are #[msg.sender]
        content is 'No available date'
      }
    }
  }
```

The initiator can decide to accept or reject a proposal and, in both cases, it sends a notification message to the responder.

```
  on ACCEPT_PROPOSAL msg {
    extract a as Action from msg
    var appointment = a.action as Appointment
    if (theAgent.isFree(appointment.fixedDate)) {
      theAgent.addMyAppointment(appointment)
      create message reply {
        performative is INFORM
      }
    } else {
      create message reply {
        performative is FAILURE
      }
    }
  }
  on REJECT_PROPOSAL msg {
    println('Responder received Refuse: ' + msg.toString())
  }
```

Finally, the `CancelAppointmentBehaviour` is shown in the following.

```
cyclic behaviour CancelAppointmentBehaviour for
     MeetingSchedulerAgent {
  on message msg
  when {
    performative is CANCEL
  } do {
    extract con as Appointment
    if (con.inviter == theAgent.AID)
      theAgent.cancelAppointment(con.fixedDate)
    else
      theAgent.removeMyAppointment(con)
  }
}
```

B. Quantitative Assessment

As shown in the previous section, JADEL implementation of the Meeting Scheduler demo is composed by the same entities of the JADE version, but it is characterized by specific syntax for agent-oriented features. Moreover, the operations and methods have the same semantics of those in JADE, but some parts are much shorter than the original ones. Beside the original demo and the JADEL implementation shown above, we obtained a third Java implementation directly by JADEL code generator. The generated code is more redundant than the original JADE demo source code, but it does not introduce significant overhead and the number of messages exchanged by the agents are the same. Hence, there is not a notable performance loss.

The scope of JADEL is mainly to simplify the use of JADE and to make clear and transparent the agent model. In order to evaluate the actual advantages in use of JADEL, we compare the original JADE implementation with our JADEL translation, by using two measures based on *Lines Of Code* (*LOC*). First, we consider the total number of non-comment and non-blank LOC of the agent, of the ontology, and of the behaviours. Then, we emphasize the fraction of domain-specific features and expressions over the total number of lines. In JADEL, such domain-specific features are those presented in Section II: entity declaration, event handler, message related expressions, and behaviours/role activation. For example, in the previously shown listing of `CancelAppointmentBehaviour`, there are 12 non-blank LOC and 6 lines of domain-specific features. In particular, those lines are the first, namely, the entity declaration, the event handling with the `on − when − do` construct, and the extraction of the received message. Thus, the fraction of domain-specific features is the 50% of the LOC. In JADE, we consider as domain-specific all the actual call to the APIs, including the lines that identify the base class, i.e., the entity.

From the Table I, we can see that JADEL implementation is shorter than JADE one. This is particularly evident when considering the ontology, that consists in only eight non-blank LOC, in JADEL. Instead, 175 lines are required by the JADE ontology, which includes also the declaration of two more classes that define concepts. Also JADEL behaviours seem to be simpler than JADE ones, with a little advantage in focusing

	MSAgent	MSOntology	Behaviours
JADEL (LOCs)	264	8	173
DS/JADEL (%)	20	75	32
JADE (LOCs)	305	175	233
DS/JADE (%)	23	11	20

Table I
NUMBER OF LOC AND PERCENTAGE OF DOMAIN-SPECIFIC (DS) FEATURES OVER THE TOTAL NUMBER OF LOC, FOR JADEL AND JADE IMPLEMENTATION OF THE MEETING SCHEDULER DEMO.

on the domain-specific tasks. The number of LOC of JADEL agent, instead, is only slightly smaller than that of JADE agent and domain-specific features are approximately the same. This is due to the point-to-point translation that has been made, in order to reproduce precisely the original demo. As a matter of fact, we have chosen not to change the demo specification, in order to make a fair comparison between the two versions. We argue that a revisited and original JADEL implementation of the Meeting Scheduler demo may be more focused on behaviours rather than on agent methods, thus improving the readability of the agent code and reducing its size.

However, LOC and domain-specific fraction of those lines are only indicators of size and density of JADEL code, but they cannot fully describe qualitative factors, such as the readability of the code, nor how much the code is reusable and maintainable.

IV. CONCLUSIONS

This paper presents a complete example of JADEL usage, related to the well-known Meeting Scheduler demo of JADE. Scope and motivations of JADEL development are briefly explained, then main abstractions, domain-specific features and expressions are introduced. The example is presented by showing JADEL source code of the Meeting Scheduler ontology, notable parts of the agent, and all the behaviours. A comparison between JADE and JADEL implementations of such a demo is made by counting non-comment and non-blank LOC and by measuring the rate of domain-specific agent-oriented features over the total number of LOC. These indicators are useful in a first approximation to enlighten some JADEL advantages, namely, its lighter syntax and the conciseness in the definition of ontologies. Nevertheless, other aspects are difficult to be precisely evaluated, due to their qualitative nature. For instance, some constructs and expressions are meant to reduce the complexity of the framework and to improve readability rather than to reduce the amount of code written. As illustrative examples, the `on – when – do` and the `extract – as` are specifically designed to clarify message reception, while the corresponding JADE patterns are repetitive and full of implementation details.

In summary, JADEL is shown to be sufficiently expressive to fully reproduce the Meeting Scheduler demo and the best results with respect to JADE are obtained in the ontology implementation. Also roles and behaviours are shorter and clearer, due to the frequent use of domain-specific constructs, especially for message passing. Moreover, the entire code of the GUI was reused, thanks to the tight integration of JADEL entities into Java type-system.

As previously said, numbers of LOCs and the percentage of DS-LOCs are not sufficient to measure the actual advantage in the use of JADEL. For this reason, future works will address the problems of language evaluation and testing. Such an evaluation could be done by scheduling the development of an application by a team of several users. Users could have different experiences in the use of JADE, from beginners to experts. Moreover, making an application that deals with a real-world problem is a good strategy for testing.

JADEL was not yet released, because it is still under development and its quality, in terms of readability and reusability, still have to be assessed. Anyway, the compiler, related documentation and examples are freely available open source upon request to authors.

REFERENCES

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

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

[3] F. Bergenti and A. Poggi, "Developing smart emergency applications with multi-agent systems," *International Journal of E-Health and Medical Communications*, pp. 1–13, 2010.

[4] S. Monica and F. Bergenti, "Location-aware JADE agents in indoor scenarios," in *Proceedings of* 16^{th} *Workshop "Dagli Oggetti agli Agenti"* (WOA '15), Napoli, Italy, 2015, pp. 103–108.

[5] S. Monica and F. Bergenti, "A comparison of accurate indoor localization of static targets via WiFi and UWB ranging," in *Advances in Intelligent Systems and Computing* (PAAMS 2016)*, Special Session on Agents and Mobile Devices (*AM*)*, 2016.

[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)*, 2012, pp. 10–15.

[7] 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*, 2015, pp. 231–246.

[8] F. Bergenti, G. Caire, and D. Gotta, "Agent-based social gaming with AMUSE," in *Procs. of the* 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, vol. 32. Elsevier, 2014, pp. 914–919.

[9] F. Bergenti, G. Caire, and D. Gotta, "An overview of the AMUSE social gaming platform," in *Procs. Workshop From Objects to Agents*, 2013.

[10] F. Bergenti and S. Monica, "Location-Aware Social Gaming with AMUSE," in *Trends in Practical Applications of Scalable Multi-Agent Systems, the PAAMS Collection* (PAAMS 2016), 2016, pp. 36–47.

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

[12] Y. Shoham, "Agent-oriented programming," *Artificial intelligence*, vol. 60, no. 1, pp. 51–92, 1993.

[13] F. Bergenti, M.-P. Gleizes, and F. Zambonelli, *Methodologies and software engineering for agent systems: the agent-oriented software engineering handbook*. Science & Business Media, 2006, vol. 11.

[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.