Overview of a Scripting Language for JADE-Based Multi-Agent Systems

Federico Bergenti, Giuseppe Petrosino
Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Università degli Studi di Parma, 43124 Parma, Italy
Email: federico.bergenti@unipr.it, giuseppe.petrosino@studenti.unipr.it

***Abstract*—This paper outlines the major features of Jadescript, a scripting language designed to support agent-oriented programming. The core abstractions that Jadescript provides are those related to event-driven agents and message passing, and the view of multi-agent systems that it promotes is closely related to that offered by JADE. Programmers using Jadescript are granted a dedicated syntax largely inspired by modern scripting languages, and major programming activities are supported in the scope of the language with a marked raise of the level of abstraction with respect to the direct use of JADE.**

I. Introduction

Agent-Oriented Programming (*AOP*), as described in [1], is a programming paradigm supported by specific programming languages, each of which provides dedicated syntax and semantics. AOP is related to the abstractions which programmers use for the construction of agents and multi-agent systems, but it is also related to the concrete syntax, and underlying semantics, that programmers adopt to manage such abstractions. Actually, the very first attempt at defining AOP (e.g., [2]) is immediately related to the definition of a programming language with suitable syntax and semantics. For some years, the widespread adoption of *agent platforms* like JADE [3] had the effect of decoupling AOP from specific programming languages because agents and multi-agent systems were mostly developed using mainstream object-oriented languages coupled with agent platforms. In recent years, the original understanding of AOP has been revitalised, and a number of AOP languages have been proposed (e.g., [4] for a list of recent proposals). This paper overviews a novel AOP language called Jadescript, which follows the path traced by its predecessor JADEL (e.g., [4], [5]), and tries to reduce even further the gap between an agent-oriented code and a semantically equivalent pseudocode (e.g., [6], [7]). This intention was the main guideline behind most of the design choices for the language. Some of such choices are described in next section to show how Jadescript was designed with a strongly expressive and easy-to-read language in mind. For this reason, Jadescript shares some characteristics with popular scripting languages like Python, e.g., collection types and the use of semantically relevant indentation.

Despite being a language intended to support programmers in the effective use of JADE, Jadescript is not an object-oriented language, at least not directly. Every Jadescript source file, if valid, is primarily intended to be compiled to one or more Java source files. Such files are then compiled to Java bytecode using one of the available Java compilers, and they are executed by a *Java Virtual Machine* (*JVM*) with the help of JADE. Such an approach has already been proved effective, and it is currently adopted, e.g., by JADEL and by SARL [8]. For the specific case of Jadescript, the choice of compiling source files to Java source files was taken mostly for the following reasons:

1) It grants interoperability between Jadescript and Java, extending the potential of Jadescript and enabling the possibility to reuse code written in one of the most popular programming languages;
2) Available Java compilers emit Java bytecode using a rich set of well-tested and decade-proven checks and optimizations, which would be pointless to try to recreate for a Jadescript to bytecode compiler; and
3) The semantics of Jadescript can be designed in terms of the underlying semantics of Java (e.g., [9]), which eases the process of designing the semantics of Jadescript and it also helps ordinary JADE and Java programmers to appreciate, and possibly adopt, Jadescript.

Finally, another important reason for the choice of compiling Jadescript source codes to Java source codes regards the availability of very powerful tools in the Eclipse ecosystem for the construction of this type of compilers. In particular, sharing the approach of, e.g., JADEL and SARL, the current implementation of the Jadescript compiler uses the tools of Xtext [10] to ensure a smooth integration with Eclipse, which ultimately ensures that Jadescript programmers are immediately supported by professional tools. Note that the minimal interface to Java which is still present in Jadescript to support integration with the features of the underlying JVM is considered low-level its use is discouraged. For example, Jadescript allows instantiating and accessing Java objects, but there is no way to define new Java classes or interfaces. Instead, concepts defined in ontologies written in Jadescript offer a way to define and manipulate structured data types, as described in next section.

This paper is organised as follows. Section II shows a brief overview of Jadescript with emphasis on supported agent-oriented abstractions using an illustrative example. Section III concludes the paper and discusses possible future developments of the language and of its tools.

II. Overview of Jadescript

Jadescript is an AOP language intended to support the implementation of agents using an event-driven style of programming with emphasis on the possibility for agents to exchange structured messages. The major abstractions that it supports are *(communication) ontologies*, *(agent) behaviours*, and *agents*. Such abstractions are well-known to JADE programmers, and Jadescript keeps them exactly as JADE programmers would expect (e.g., [11] for a detailed description of such abstractions). This section shows how such abstractions are supported in Jadescript together with a description of other features that the language provides to accommodate ordinary abstractions of event-driven programming. Note that the overview of Jadescript discussed in this section is by far not exhaustive and many features of the language are not described. Only a selected set of features is presented and interested readers are directed to the documentation which comes with the distribution of Jadescript tools for further discussions and examples.

In order to ease descriptions, the features of Jadescript discussed in this section are often presented using illustrative examples. Besides minimal examples used in the description of statements and expressions, the illustrative example used to present the agent-oriented features of the language is a variation of the well-known *ping-pong example* discussed in JADE documentation [11]. The proposed variation assumes that there are two types of agents: agents of type *ping* and agents of type *pong*. Ping agents contact known pong agents with direct messages, and pong agents reply to such messages. Ping agents count sent messages and include the current value of their counts in outbound messages. Pong agents reply to each received message citing the value of the count included in the message. In detail, ping agents send FIPA request messages [12] to all known pong agents to ask them to perform an action called `reply` with argument `counter`, which is the current value of the message count of the sender. Upon executing the requested action, pong agents send back a FIPA inform message [12] to the sender of the message using the proposition `alive(counter)` as message content. The multi-agent system is assumed to be hosted in a JADE platform, and the list of pong agents is passed to ping agents upon initialisation.

The remaining of this section describes the features that can be included in a Jadescript source file. Note that a Jadescript source file always has a specific structure. It starts with the mandatory declaration of the module where all features defined in the source file are contained using the keyword `module`. Modules are simply intended as named groups of features, each of which can be public or private to the module. After the declaration of the module, imported features from other modules are enumerated using the keyword `import`. Finally, a list of definitions related to the major features supported by the language, i.e., ontologies, behaviours, and agents, is provided. All such features are discussed in the following after a brief note on the procedural features of the language.

A. Data Types

Jadescript is a statically typed language, and its type system is based on five groups of types: *primitive* types, *ontology* types, *collection* types, *behaviour* types, and *agent* types. The following is the list of primitive types currently supported by the language (in alphabetical order):

1) `boolean`: Boolean logical values;
2) `double`: double-precision floating-point numbers;
3) `float`: single-precision floating-point numbers;
4) `integer`: integer numbers; and
5) `text`: texts (strings of characters).

Note that primitive types, with the exception of texts, are mapped to Java primitive types with similar names.

Structured types are supported in Jadescript in terms of concepts, propositions, predicates and actions declared in ontologies (see Subsection II-C). Such structured types can refer to other primitive, collection or ontology types. Fig. 1 shows an example of a declaration of an ontology which includes the declarations of two concepts, one predicate, and one action.

```
ontology BookShop
  concept person(surname as text,
    name as text)

  concept book(author as person,
    title as text, price as double)

  predicate authorOf(author as person,
    item as book)

  action sell(customer as person,
    item as book)
```

Fig. 1. Example of an ontology written in Jadescript.

Jadescript offers two collection types used to refer to two of the most common data structures, namely lists and maps. Examples of collection types are `list of integer` and `map of integer : text`. Collection types can be composed and with any available data type. In particular, any data type can be used to declare the type of the keys of a map.

Finally, behaviour types and agent types are provided for the manipulation of behaviours (see Subsection II-D) and of agents (see Subsection II-E), respectively. Every behaviour definition or agent definition implicitly defines a new data type, whose use is restricted with respect of other data types. In particular, Jadescript provides specific constructs for the manipulation of behaviours and of agents to ensure that such abstractions are given first-class support in the language.

Given that Jadescript is designed as a statically typed language, the types of all features defined and referenced in a source code are known at compile time, and there is no way to synthesise new types at run time. However, in order to relieve the programmer from the burden of explicitly declaring

the types of variables, the Jadescript compiler infers automatically the types of variables from mandatory initialisation expressions. Therefore, there is no need to explicitly declare the types of variables in Jadescript. In addition, the compiler tracks declared names and when a new name is found in an assignment statement, it assumes that a new variable is implicitly declared. Note that the sort of type inference that Jadescript provides is limited with respect to other languages, and the types of some elements of a Jadescript source code, e.g., the types of formal parameters in function declarations, need to be explicitly stated.

B. Statements and Expressions

Jadescript is an event-driven language and, as such, it provides the common statements and expressions of procedural languages. The following is a summary of supported statements and expressions that cannot be considered specific to the agent-oriented style of programming. AOP statements and expressions are discussed together with the agent-oriented abstractions that they support.

In the tradition of the programming languages derived from the C language, function calls can be used as statements in Jadescript. In addition, the `do-nothing` statement is just a placeholder to allow users to write empty blocks of code. The `do-nothing` statement is necessary because the delimitation of blocks in Jadescript is based on indentation.

Jadescript provides a single statement to declare variables and to perform assignments. The common infix = operator is provided for such tasks. As discussed at the end of previous subsection, the types of variables is inferred by the compiler from mandatory initialisation expressions, and there is no need to make types explicit.

Jadescript provides the `create` statement to create instances of collection and ontology types. After the `create` keyword, the first required term is the type of which the instance is being created. Right after that, an identifier is required, which determines the name of the new variable that refers to the created value. If needed, a list of named arguments is provided after the `with` keyword. Fig. 2 is an example which uses the ontology types declared in Fig. 1.

```
1 create book odyssey with author homer,
2   title "Odyssey", price 19.99
```

Fig. 2. Example of the use of the `create` statement in Jadescript.

The classic `if` statement can be used to express conditional blocks of code. Like in most procedural languages, the `if` statement can also have multiple `if-else` branches and an optional `else` branch at the end. After every condition, a new block of code must be opened by increasing the indentation level. The semantics of this statement is intuitively the same of conditional constructs in common procedural languages.

Jadescript provides a first form of iterative statement in terms of the common `while-do` statement. In addition, iteration over collections is supported with the `for-in-do` statement, which can be used to iterate over the elements of lists or over the keys of maps. At each iteration, an element from the collection which follows the `in` keyword is extracted and, before the body of the statement is executed, it is assigned to a variable whose name is declared after the `for` keyword.

In the procedural parts of a Jadescript source code, the `return` statement can be used to terminate the execution of a procedure or of a function and possibly return a value to the caller. The use of the `return` statement is mandatory in functions and all possible execution paths of a function must terminate with a proper `return` statement.

A set of statements is provided by Jadescript to work on collection types. In particular, `add-to`, `remove-from`, and `clear` statements are used to manipulate the contents of lists. Note that `add-to` and `remove-from` can possibly specify an optional index to work on an element different from the last element of the list. Similarly, `remove-from` and `clear` are available for maps. Note that other common operations on collections, e.g., the operation to access singles elements of a list, are provided in terms of expressions.

Just like common procedural languages, Jadescript offers a system of expressions. Every expression computes a value and the types of computed values are determined by the compiler. Operations can be combined to share operands and common precedence and associativity rules are adopted to disambiguate the order of evaluation. As usual, a limited set of expressions can also be used at the left side of the = operator.

Ordinary Boolean expressions are formed in Jadescript using the keywords `and`, `or`, and `not`. Such Boolean operations follow a lazy evaluation scheme: if the value of the whole operation can be deducted from the evaluation of the first operand, the second operand is not evaluated. In addition, ordinary comparison operators with common semantics are provided. Finally, arithmetic expressions are supported in Jadescript using ordinary operators for addition, subtraction, multiplication, division, and modulo.

Jadescript provides operators to work with collection types. Ordinary square brackets are available to access the elements of lists and maps. When applied to lists, they require a nonnegative integer to be used as an index in the list. When applied to maps, they require a value compatible with the type of the keys to access the corresponding value in the map. Note that square brackets can also be used at the left side of the = operator to modify lists and maps. In addition, the `size-of` operator can be used to retrieve the number of elements of lists and maps. The `contains` operator has the function to check if the collection returned by the evaluation of the left-side operand contains the element returned by the evaluation of the right-side operand. It can work on lists and on the keys of maps. Finally, Jadescript offers a concise way to instantiate lists and maps by declaring their contents. A list literal is written as a comma-separated list of values between square brackets. The type of the elements contained in the list is computed by finding the closest common ancestor of all types in the list literal. Similarly, Jadescript offers a way to quickly instantiate a map by enumerating a set of key-value pairs. A

map literal is written as a comma-separated list of key-value pairs between curly brackets, where keys are separated from values by colons. The type of the keys of the map is computed by finding the closest common ancestor of all types of the keys of the map literal. The type of the values of the map is computed by finding the closest common ancestor of all types of the values of the map literal.

Type casting is supported in Jadescript using the `as` operator, which forces the type of the result of the evaluation of the expression on its left side to the type specified on its right side, if types are actually compatible. Type inspection is supported in Jadescript by means of a specific operator. The `is` operator checks if the type of the value computed by the expression at its left side is actually compatible with the type specified as right operand.

C. Ontologies

Ontologies are used by agents to share concepts, actions, predicates, and propositions, and to refer to them in messages (e.g., [13]). The support for ontologies that Jadescript provides allows managing all such features. Concepts are structured entities used to describe the world where agents live. A concept is defined by stating its name, its main properties, and whether or not it can be considered an extension of another concept. Actions are structured entities used to refer to the actions that agents can be requested to perform. An action is stated by declaring its name, its main properties, and whether or not it can be considered an extension of another action. Predicates are structured entities used to state logic expressions about the world where agents live. Like concepts and actions, they can have a set of properties, and they can extend other predicates. Similarly to predicates, propositions are logic expressions: they serve the same function as predicates, but they have no properties. Finally, besides the declaration of concepts, actions, predicates, and propositions, an ontology is characterised by a name, unique in its module, and by an optional base ontology that it extends. Fig. 3 shows the declaration of the ontology used to implement the ping-pong example in Jadescript.

```
1 ontology PingPong
2     action reply(counter as integer)
3
4     predicate alive(counter as integer)
```

Fig. 3. Jadescript ontology used in the ping-pong example.

With the exception of propositions, all features declared in an ontology can be used to create structured values, whose elements can be accessed using the `of` operator. Fig. 4 shows an example of the use of such an operator at the left side and at the right side of the = operator.

```
1 author of odyssey = author of iliad
```

Fig. 4. Example of the use of the `of` operator in Jadescript.

D. Behaviours

Behaviours are used to describe how agents operate during their lifetime in the multi-agent system. In Jadescript, these are built on top of JADE behaviours and they are characterised by the peculiar scheduling mechanisms of JADE behaviours [11]. The `behaviour` construct can be used to define new Jadescript behaviours. A minimal behaviour is declared by stating its name, which must be unique in the module, and its type, which can be `cyclic` or `one shot`. More complex types of behaviours, as supported by JADE, are planned for future versions of the language. The declaration of a behaviour can restrict the type of the agents that can use the behaviour by means of the construct `for-agent`, and it can link the behaviour to an ontology with the construct `uses-ontology`. Finally, the declaration of a behaviour is completed with a list of optional features to be used to actually implement the behaviour. Such features are properties, functions, procedures, actions, and (event) handlers. Note that, whenever an expression can be used in the definition of one of such features, the keyword `agent` can be used to refer to the agent which is currently linked with the behaviour.

A property of a behaviour is a part of the run-time state of the behaviour. It is distinguished by a name, unique in the declaration of the behaviour, a type, and an initialisation expression. The type is deducted at compile-time from the type of the initialisation expression by the compiler, and it is normally not explicitly stated. Properties can be accessed from other features of the behaviour, and also from other agents and behaviours, using the `of` operator.

Functions and procedures can be declared in behaviours to define parameterized blocks of code to perform tasks. The declarations of functions must specify the types of the values that functions return. Functions and procedures can have zero or more arguments. The names and the types of such arguments must be specified with a list of formal parameters. Finally, functions and procedures have bodies in which sequences of statements define what they are supposed to do during their executions.

Actions are the primary features used to describe how behaviours perform their tasks. When a behaviour is selected for execution, its action is immediately executed. Note that actions declared in behaviours are conceptually different from actions declared in ontologies. The former are executable pieces of code scheduled for execution with the behaviour, while the latter are descriptions of the actions that agents can be requested to perform, and they are primarily intended to support communication among agents.

Various `on` constructs can be used to declare (event) handlers in the scope of behaviours. Handlers are used to identify the blocks of code that are supposed to be executed when interesting events occur. For the time being, the most important type of event handler available in behaviours is intended to support the reception of messages. Handlers of this type assign a name to the messages being received and specify a condition that interesting messages are demanded to satisfy.

Behaviours can be activated using the specific `activate-behaviour` statement, which can be used inside the actions of agents and behaviours. This statement creates a new behaviour and it marks the behaviour as active, so that the behaviour can be scheduled for execution by the agent. Some behaviours need a set of arguments in order to be properly initialised. Such arguments are passed to the behaviour by using the `with` keyword in the scope of the `activate-behaviour` statement.

It is evident that sending and receiving messages are the central activities to support agent communication in multi-agent systems. The `on-when-do` construct is meant to manage incoming messages, while the `send` statement is used to manage outgoing messages. There are two syntactical forms of the `send` statement. The most basic form can be used if an instance of the concept describing FIPA messages is available, and it allows accessing all the features of FIPA messages [12]. On the contrary, the simplified form is an alternative way to send messages in a more concise manner. The simplified `send` statement creates a message and sends it, but only the performative, the list of receivers, the content, and, implicitly, the ontology can be specified.

Fig. 5 shows two behaviours used to implement the ping-pong example. The `SendToPong` behaviour is scheduled by ping agents to send messages to pong agents, while `ReplyToPings` behaviour is used by pong agents to reply. The count of outbound messages is a property of ping agents.

```
1  one shot behaviour SendToPong for
2    agent Ping uses ontology PingPong
3
4    other as aid
5
6    on create with pong as aid do
7      other = pong
8
9    do
10     counter = counter of agent
11
12     send request reply(counter) to other
13
14     counter of agent = counter + 1
15
16 cyclic behaviour ReplyToPings for
17   agent Pong uses ontology PingPong
18
19   on message m when
20     performative of m is request and
21     counter of m is reply do
22     r = content of m as reply
23     c = counter of r
24
25     send inform alive(c) to sender of m
```

Fig. 5. Two Jadescript behaviours used in the ping-pong example.

E. Agents

Jadescript agents are the core abstraction used to build Jadescript multi-agent systems. Essentially, they are JADE agents and they can be defined using the `agent` construct. Agents are structured in terms of the following features: properties, procedures, functions, and (event) handlers. Properties, procedures, and functions are the same features available in the declaration of behaviours. In particular, the `of` operator can be used to access the properties of agents. Handlers are restricted forms of handlers with respect to the handlers available in behaviours because they can be used only to react to events regarding the lifecycle state of agents. One of such events is captured with the `on-create` handler, which is activated just before the agent becomes available to the multi-agent system. Similarly, the `on-destroy` handler is triggered just before the agent is removed from the multi-agent system. Interested readers should consult JADE documentation [11] for a description of the possible lifecycle states of an agent. Fig. 6 shows the agents used to implement the ping-pong example. Note that the example also uses a `ReplyToPongs` behaviour, not shown in the figure, to ensure that ping agents would iteratively send messages to pong agents.

```
1  agent Pong
2    on create
3      activate behaviour ReplyToPings
4
5  agent Ping
6    counter = 1
7
8    on create with args as list of text do
9      activate behaviour ReplyToPongs
10
11     for t in args do
12       activate behaviour SendToPong with
13         pong = aid(t)
```

Fig. 6. Jadescript ping and pong agents.

III. Conclusion

This paper presented a brief overview of the Jadescript programming language. Jadescript is a language for event-driven programming which supports the implementation of agents and multi-agent systems by means of specific syntax and semantics. The adopted syntax is designed to reduce the gap between a Jadescript agent and a semantically equivalent pseudocode, and it uses the common features of a modern scripting language. The long-term vision of this project is to allow programmers to easily adopt agents and multi-agent systems to benefit from their relevant features in terms of reusability and composability, and interoperability (e.g., [14]).

One of the major planned developments of the discussed work regards the possibility of managing events different from the reception of messages. This would require the definition

of a framework to let event provider feed events to agents, and it would require syntactic enhancements to the language to allow reacting to events in the environment and proactively sensing the environment. Such a possibility would allow the use of Jadescript in situations where agents are immersed in complex and dynamic environments, e.g., the industrial environments discussed in [15]–[18] or the highly dynamic environments discussed in [19]. Furthermore, the results of previous projects suggest that accommodating generic events as first-class citizens of the language would make Jadescript a valid tool for applications related to e-health (e.g., [20]), next-generation enterprise collaboration (e.g., [21], [22]), and indoor navigation (e.g., [23]–[30]).

Jadescript is currently supported by a set of tools packed as an Eclipse plug-in to ease the adoption from programmers with no specific background on agent technology. The current version of tools and related documentation is available upon request for authors, and an open-source distribution is planned for the near future.

REFERENCES

[1] Y. Shoham, "An overview of agent-oriented programming," in *Software Agents*, J. Bradshaw, Ed., vol. 4. MIT Press, 1997, pp. 271–290.

[2] Y. Shoham, "AGENT-0: A simple agent language and its interpreter," in *Proc. 9^{th} Nat. Conf. Artificial Intelligence (AAAI 1991)*, vol. 91, 1991, pp. 704–709.

[3] 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 International Publishing, 2005, pp. 125–147.

[4] F. Bergenti, E. Iotti, S. Monica, and A. Poggi, "Agent-oriented model-driven development for JADE with the JADEL programming language," *Computer Languages, Systems & Structures*, vol. 50, pp. 142–158, 2017.

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

[6] F. Bergenti, E. Iotti, S. Monica, and A. Poggi, "A case study of the JADEL programming language," in *Proc. 17^{th} Workshop "From Objects to Agents"*, ser. CEUR Workshop Proceedings, vol. 1664. RWTH Aachen, 2016, pp. 85–90.

[7] F. Bergenti, E. Iotti, S. Monica, and A. Poggi, "A comparison between asynchronous backtracking pseudocode and its JADEL implementation," in *Proc. 9^{th} International Conference on Agents and Artificial Intelligence (ICAART 2017)*, vol. 2. SciTePress, 2017, pp. 250–258.

[8] S. Rodriguez, N. Gaud, and S. Galland, "SARL: A general-purpose agent-oriented programming language," in *2014 IEEE/WIC/ACM Int. Conf. Intelligent Agent Technology*. Warsaw, Poland: IEEE Computer Society Press, 2014.

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

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

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

[12] Foundation for Intelligent Physical Agents, "FIPA specifications," 2002, available at http://www.fipa.org/specifications.

[13] M. Tomaiuolo, P. Turci, F. Bergenti, and A. Poggi, "An ontology support for semantic aware agents," in *Proc. 7^{th} Int. Bi-Conf. Workshop Agent-Oriented Information Systems III (*AOIS 2005*)*, ser. LNAI, vol. 3529. Springer International Publishing, 2006, pp. 140–153.

[14] F. Bergenti, "A discussion of two major benefits of using agents in software development," in *Engineering Societies in the Agents World III: 3^{rd} Int. Workshop ESAW 2002*, P. Petta, R. Tolksdorf, and F. Zambonelli, Eds. Springer International Publishing, 2003, pp. 1–12.

[15] S. Monica and G. Ferrari, "Optimized anchors placement: An analytical approach in UWB-based TDOA localization," in *Proc. 9^{th} International Wireless Communications & Mobile Computing Conference* (IWCMC 2013). Cagliari, Italy: IEEE, 2013, pp. 982–987.

[16] S. Monica and G. Ferrari, "Impact of the number of beacons in PSO-based auto-localization in UWB networks," in *Proc. European Conference on the Applications of Evolutionary Computation* (EvoApplications 2013), ser. LNCS, vol. 7835. Springer, 2013, pp. 42–51.

[17] S. Monica and F. Bergenti, "A comparison of accurate indoor localization of static targets via WiFi and UWB ranging," in *PAAMS 2016: Trends in Practical Applications of Scalable Multi-Agent Systems*, ser. AISC, vol. 473. Springer, 2016, pp. 111–123.

[18] S. Monica and G. Ferrari, "Low-complexity UWB-based collision avoidance system for automated guided vehicles," *ICT Express*, vol. 2, pp. 53–56, 2016.

[19] 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} Int. Conf. PAAMS 2016*, Y. Demazeau, T. Ito, J. Bajo, and M. J. Escalona, Eds. Springer International Publishing, 2016, pp. 36–47.

[20] A. Poggi and F. Bergenti, "Developing smart emergency applications with multi-agent systems," *Int. J. E-Health and Medical Communications*, vol. 1, no. 4, pp. 1–13, 2010.

[21] F. Bergenti, E. Franchi, and A. Poggi, "Agent-based social networks for enterprise collaboration," in *Proc. 20^{th} IEEE Int. Conf. Enabling Technologies: Infrastructure for Collaborative Enterprises* (WETICE 2011). IEEE Press, 2011, pp. 25–28.

[22] F. Bergenti, G. Caire, and D. Gotta, "An overview of the AMUSE social gaming platform," in *Proc. Workshop "From Objects to Agents"*, ser. CEUR Workshop Proceedings, vol. 1099. RWTH Aachen, 2013.

[23] S. Monica and G. Ferrari, "Particle swarm optimization for auto-localization of nodes in wireless sensor networks," in *Proc. 11^{th} Int. Conf. Adaptive and Natural Computing Algorithms (*ICANNGA 2013*)*, ser. LNCS, vol. 7824. Springer International Publishing, 2013, pp. 456–465.

[24] S. Monica and G. Ferrari, "Accurate indoor localization with UWB wireless sensor networks," in *Proc. 23^{rd} IEEE Int. Conf. Enabling Technologies: Infrastructure for Collaborative Enterprises* (WETICE 2014). IEEE Press, 2014, pp. 287–289.

[25] S. Monica and F. Bergenti, "Location-aware JADE agents in indoor scenarios," in *Proc. 16^{th} Workshop "From Objects to Agents"*, ser. CEUR Workshop Proceedings, vol. 1382. RWTH Aachen, 2015, pp. 103–108.

[26] S. Monica and F. Bergenti, "Optimization based robust localization of JADE agents in indoor environments," in *Proc. 3^{rd} Italian Workshop on Artificial Intelligence for Ambient Assisted Living (AI*AAL.IT 2017)*, ser. CEUR Workshop Proceedings, vol. 2061. RWTH Aachen, 2017, pp. 58–73.

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

[28] S. Monica and F. Bergenti, "Indoor localization of JADE agents without a dedicated infrastructure," in *MATES 2017: Multiagent System Technologies*, ser. LNCS, vol. 10413. Springer, 2017, pp. 256–271.

[29] S. Monica and F. Bergenti, "An experimental evaluation of agent-based indoor localization," in *Proc. Computing Conference 2017*. IEEE, 2018, pp. 638–646.

[30] S. Monica and F. Bergenti, "An optimization-based algorithm for indoor localization of JADE agents," in *Proc. 18^{th} Workshop "From Objects to Agents"*, ser. CEUR Workshop Proceedings, vol. 1867. RWTH Aachen, 2017, pp. 65–70.